东南大学算法设计与分析课程考试复习试题题库及答案

1 什么是基本运算？

答：基本运算是解决问题时占支配地位的运算（一般 1 种，偶尔两种）；讨论一个算法优劣时，只讨论基本运算的执行次数。

2 什么是算法的时间复杂性（度）？

答：算法的时间复杂性（度）是指用输入规模的某个函数来表示算法的基本运算量。T(n)=4n3

3 什么是算法的渐近时间复杂性？

答：当输入规模趋向于极限情形时（相当大）的时间复杂性。

4 表示渐进时间复杂性的三个记号的具体定义是什么？

答：1. T(n)= O(f(n))：若存在 c ＞ 0，和正整数 n0≥1，使得当 n≥n0 时，总有 T(n)≤c*f(n)。 （给出了算法时间复杂度的上界，不可能比 c*f(n)更大）

2. T(n)=Ω(f(n))：若存在 c ＞ 0，和正整数 n0≥1，使得当 n≥n0 时，存在无穷多个 n ，使得 T(n)≥c*f(n)成立。（给出了算法时间复杂度的下界，复杂度不可能比 c*f(n)更小）

3. T(n)= Θ(f(n))：若存在 c1,c2>0，和正整数 n0≥1，使得当 n≥n0 时， 总有 T(n)≤c1*f(n)， 且有无穷多个 n，使得 T(n)≥c2*f(n)成立， 即：T(n)= O(f(n))与 T(n)=Ω(f(n))都成立。（既给出了算法时间复杂度的上界，也给出了下界）

5 什么是最坏情况时间复杂性？什么是平均情况时间复杂性？

答：最坏情况时间复杂性是规模为 n 的所有输入中，基本运算执行次数为最多的时间复杂性。

平均情况时间复杂性是规模为 n 的所有输入的算法时间复杂度的平均值（一般均假设每种输入情况以等概率出现）。

6 一般认为什么是算法？什么是计算过程？

答：一般认为，算法是由若干条指令组成的有穷序列，有五个特性 a.确定性（无二义）b.能行性（每条指令能够执行）c.输入 d.输出 e.有穷性（每条指令执行的次数有穷）

只满足前 4 条而不满足第 5 条的有穷指令序列通常称之为计算过程。

7 算法研究有哪几个主要步骤？主要从哪几个方面评价算法？

答：算法研究的主要步骤是 1)设计 2）表示 3）确认，合法输入和不合法输入的处理 4）分

析 5）测试

评价算法的标准有 1）正确性 2）健壮性 3）简单性 4）高效性 5）最优性

8 关于多项式时间与指数时间有什么样的结论？

答：1. 多项式时间的算法互相之间虽有差距，一般可以接受。

2. 指数量级时间的算法对于较大的 n 无实用价值。

9 什么是相互独立的函数序列？何时称函数项 μk(x)能被其它函数项线性表出？

答：设{μ0(x)，μ1(x)，μ2(x)，...，μn(x)，...}是某一数域上的函数序列，（x 的值以及 μk(x)(k=0,1,2，…)的值都在同一个数域中）任取 μk(x)（k=0,1,2，…)，不存在数域中的数 α1，α2,…，αp，使得 μk(x) = α1μi1(x) + α2μi2 (x) + … + αpμip (x) ，即任何一个函数项 μk(x)不能被其它函数项线性表出。

10 根据特征根的情况，常系数线性递归方程的解有哪几种不同的形式？

答：1.若方程(**)恰有 r 个互不相同的特征根 α1，α2，…，αr （即 i≠j 时有 αi≠αj)，则齐次方程(*)的解为 an＝A1+ A2+ …+Ar(齐通解，即齐次方程的通解）（A1～Ar 为待定系数，可由 r 个连续的边界条件唯一确定）

2. 若 α1，α2 是(**)方程的一对共扼复数根 ρ 和 ρ， eiθeiθ.则这两个根对应的解的部分为 Aρncos(nθ)+Bρnsin(nθ) (A,B 为实的待定系数)

3. 若 α 是(**)方程的 k 重根，则 α 对应的解的部分为 C1αn+C2nαn+C3n2αn+ …+Ck nk-1αn (C1～Ck 为待定常数)

4. 若(*)方程中的 f(n)≠0(非齐次)，且 q(n)是(*)的一个解， 则(*)方程的解为： (*)的齐通解（含有待定系数）+q(n) （非齐特解）， （齐通解中的待定系数由边界条件唯一确定）

11 求和中的通项与积分中的被积函数之间有什么样的关系？

答：求和中的通项的表达形式一般就是被积函数，一般用放缩的方法求得通项得上下界。

12 主定理的内容是什么？根据主定理的结论，可以获得哪些关于算法改进的启示？

答：T(n)=a*T(n/b)+f(n)

1) 若有 ε＞0，使 f(n)=0()（即 f(n)的量级多项式地小于的量级），则 T(n)= Θ ()。

2) 若 f(n)= Θ ()（即 f(n)的量级等于的量级），则 T(n) =Θ ()。

3) 若 f(n)= Θ ()，则 T(n)=Θ ( )

3) 若有 ε >0，使 f(n)=**Ω**() ε +)(aLogbn(即 f(n)的量级多项式地大于的量级），且满足正规性条件： abLogn 存在常数 c<1，使得对所有足够大的 n，有 a*f(n/b)≤c*f(n)，则 T(n)=Θ (f(n))。

正规性条件的直观含义：对所有足够大的 n，a 个子问题的分解准备与再综合所需要的 时间总和，严格小于原问题的分解准备和综合所需要的时间。因此一般来说，对于时间复杂度满足递归关系 T(n)=a*T(n/b)+f(n)的算法，只需比较 f(n)与的量级大小，aLogbn 算法的时间复杂度总是与量级大的那个相同（即小的那个可以忽略）；若 f(n)与的量级相同（或只差）， aLogbnnLogk 则算法的时间复杂度为 f(n)*。

主定理对于算法改进得启示在于：

1）当 T(n)= Θ ()时，即当的量级高于 f(n)的量级时，n 的量级在算法的时间复杂度中起主导作用。因而此时首先应当考虑使 aLogbnaLogbnaLogb logb a 的值变小，而暂时无须考虑如何改善 f(n)，即此时要考虑减小 a （子问题个数）且使 b 不变（n/b 为子问题规模），或增大 b（减小子问题规模）而使 a 不变。也就是说，此时的重点应考虑改进子问题的分解方法，暂不必考虑改进子问题组合时的处理。

2）当 f(n)的量级高于或等于的量级时，则 f(n)的量级在算法的时间复杂度中起主导作用。因而此时首先应当考虑把 f(n)的量级往下降，即此时应着重改善子问题组合时的处理方法，减少该部分工作的处理时间 f(n)。

13 分治法的要领是什么？（分治法可分为哪三个主要步骤？）

答：分治法的要领

分治法是把一个规模较大的问题分解为若干个规模较小的子问题，这些子问题相互独立且与原问题同类； 首先求出这些子问题的解，然后把这些子问题的解组合起来得到原问题的解。由于子问题与原问题是同类的，故使用分治法很自然地要用到递归。

因此分治法分三步：

1 将原问题分解为子问题（Divide）

2 求解子问题（Conquer）

3 组合子问题的解得到原问题的解（Combine）

14 分治法求最大、最小元算法的主要思想？

答：当 n=2 时，一次比较就可以找出两个数据元素的最大元和最小元。 当 n>2 时，可以把 n 个数据元素分为大致相等的两半， 一半有 n/2 个数据元素，而另一半有 n/2 个数据元素。 先分别找出各自组中的最大元和最小元，然后 将两个最大元进行比较，就可得 n 个元素的最大元； 将两个最小元进行比较，就可得 n 个元素的最小元。

求最大、最小元算法的时间复杂度（比较次数）下界是多少？分治算法在什么情况下可以达到下界？

答：在规模为 n 的数据元素集合中找出最大元和最小元， 至少需要 3n/2-2 次比较，即 3n/2-2 是找最大最小元算法的下界。当 n=2k,或当 n≠2k 时，若 n 是若干 2 的整数幂之和，（e.g. 42=32+8+2），

则算法的时间复杂度仍可达到 3n/2-2。

15 如何用分治法求两个 n 位二进制数 x 和 y 的乘积？算法的时间复杂度是多少？

答：若 n=2k，则 x 可表为 a2n/2+b，y 可表为 c2n/2+d（如图），其中 a，b，c，d 均为 n/2 位二进制数。 于是 x*y= (a2n/2+b) (c2n/2+d) = ac2n + (ad+bc)2n/2 + bd，计算式 ac2n + (ad+bc)2n/2 + bd 中的 ad+bc 可写为： 而 (ad+bc)= (a+b) (d+c) - ac - bd，因此在 ac 和 bd 计算出之后， 只要再做 4 次加减法，一次（n/2 位数的）乘法就可以计算出 ad+bc。 而原来计算(ad+bc)需要做 2 次乘法、一次加法； 新的计算公式比原方法少做了一次乘法。 T(n)=3T(n/2) + Θ(n)，即 a=3， b=2， f(n)=Θ(n)。 此时有：= = naLogbn32Logn1.59，并仍有 f(n)=0()， ε -)(aLogbn 于是有 T(n)= Θ(n1.59)，比 Θ(n2)要好不少。

16 用 200 字概括 Select（求第 k 小元）算法的主要思路。

答：1.若 S<50，则采用堆排序的方法找出第 k 小的元素

2.将 n 个元素分成[n/5]组，每组 5 个元素

3.对每个 5 元组进行排序，全部 5 元组排序后，从每组中取出第 3 个元素（中间元）得到一个长为[n/5]的数组 M

4.递归调用 Select([|M|/2],M)，即在 M 数组中找到第[|M|/2]小的数（中位数），记为 m

5.依次扫描整个数组 S，此项工作所需时间为 O(n)。

当 si<m 时将 si 放入数组 S1；

当 si=m 时将 si 放入数组 S2；

当 si>m 时将 si 放入数组 S3；

在得到的 3 个集合中，S1 中的数均小于 m；

S2 中的数均等于 m；S3 中的数均大于 m。

6. 按照 k 值大小，共可分成下列三种情况（注意 S2 至少有一个元素 m）： k≤|S1|；|S1|<k≤|S1|+|S2|；以及 k>|S1|+|S2|。 下面针对这三种情况分别进行讨论。

6.a:若 k≤|S1|,则第 k 小元素必定在 S1 中。此时递归调用 Select(k,S1)，就可以获得第 k 小元素。 因大于等于 m 的数据元素至少有 3n/10-6 个， 而 S1 中的数均小于 m，故 S1 中的数据元素至多有 7n/10+6 个， 即|S1|≤7n/10+6。因此，调用 Select（k,S1）的 时间复杂度不超过 T(7n/10+6)。

6.b：若|S1|<k≤|S1|+|S2|，则第 k 小元素必定在 S2 中。 因为 S2 中的元素的值均等于 m，故第 k 小元素就是 m。 由于答案已经得到，此时立刻返回第 k 小元素 m。 这部分工作的时间复杂度为 O(1)。

6.c：若 k>|S1|+|S2|，则第 k 小元素必定大于 m，因此在 S3 中。 而且此时该元素在 S3 中应为第 k-|S1|-|S2|小的元素。 于是递归调用 Select（k-|S1|-|S2|, S3）， 就可以获得 S 中的第 k 小元素。 因小于等于 m 的数据元素至少有 3n/10-6 个

1 的情况下，T(n)分别为什么？≥1 及 p+q<时间复杂度为 T(n)=T(p*n)+T(q*n)+a*n 时，在 p+q

答：T(n)≤a*n*∑k=0∞(p+q)k

17 矩阵相乘算法目前最好的时间复杂度是多少？

答：目前矩阵乘法最好的时间复杂度是能做到 O（n2.376）

18 叙述 Strassen 矩阵相乘算法的主要思路和意义。

答：把矩阵 A，B 分成 4 个规模为 n/2 的子矩阵快

C11=A11B11+A12B21, C12=A11B12+A12

C21=A21B11+A22B21, C22=A21B12+A22B22

同时引入下列 Mi(i=1,2...7)

则计算两个 n 阶矩阵的乘法为 7 对 n/2 阶矩阵的乘法(时间为 7T(n/2)),以及 18 对 n/2 阶矩阵的加减法则递归方程为 T(n)=7T(n/2)+ Θ(n2),由主定理得 T(n)= Θ(n2.81). Strassen 矩阵相乘算法意义在于打破了人们认为矩阵乘法得时间复杂度为 Θ（n3）得固定看法。

19 试用 200～300 字概述寻找最近点对算法的主要步骤。该算法中有哪几点最为关键？该算法是否可改进？

答：主程序算法：

读入 n 个点的坐标,这 n 个点的 x 坐标和 y 坐标 分别放在 X,Y 两个数组中,然后进行预处理： 对 X 数组中的 n 个 x 坐标值按从小到大的次序进行排序， 排序过程中保持 x 坐标和 y 坐标的对应关系：

若 X[i]与 X[j]对换位置，则 Y[i]与 Y[j]也做相同的对换。 另外，若两个点的 x 坐标相同，则 y 坐标值小的排前。 X 数组排好之后就固定了，以后不再改变， 以便在 O(1)时间对其实现分拆。（排序时间为 Θ(n log n)） 将数组 IND 初始化为：IND[i]=i（i=1,2,----,n）。 数组 IND 即是用来保持 x 坐标和 y 坐标的对应关系的机制， IND[i]记录的是 其 y 坐标值为 Y[i]的点所对应的 x 坐标在 X 数组中的下标。 对 Y 数组中的 n 个 y 坐标值按从小到大的次序进行排序， 排序过程中保持 y 坐标和 x 坐标的对应关系： 若 Y[i]与 Y[j]对换位置，则 IND [i]与 IND [j]也做相同的对换。 这样，当给了一个点的 y 坐标 Y[i]之后， 就可以在 O(1)时间找到其对应的 x 坐标： Y[i]与 X[IND [i]]就是该点的 y 坐标和 x 坐标。

调用子程序 FCPP(1,n,X,Y,IND, δ,p,q) 就可求得 n 个点中的最近点对(p,q)和最小距离 δ。 子程序 FCPP 的主要执行过程： 首先看当前处理的点数。若不超过 3 个点，就直接进行相互比较。 若超过 3 个点，则把点的 y 坐标分为两部分：左边和右边。

然后进行分治，求得两边的 δL 和 δR，从而求得 δ。 求出分割线，扫描当前的所有点，把落到 2δ 带状区域内的点找出来， 并使这些点的 y 坐标仍然保持从小到大的次序。对落到 2δ 带状区域内的每一个点检查其后面的 7 个点，若有距离更近的点对，则把最小距离 δ （及最近点对(p,q)）更新， 执行完毕时，最小距离 δ 及最近点对(p,q)就得到了。

子程序 FCPP(j,k,X,Ypres,INDpres, δ,p,q)中的参数说明： X 数组存放已排好序的 n 个点的 x 坐标。 j,k 为当前处理的 X 数组一段中的最小和最大下标。Ypres 数组存放当前处理的 k-j+1 个点的 y 坐标 （已按从小到大的次序排好）。INDpres 数组的长度也是 k-j+1，INDpres[i]记录了其 y 坐标值 为 Ypres [i]的点的 x 坐标在 X 数组中的下标值。 δ,p,q 均为返回值， 给出当前处理的 k-j+1 个点中的最小距离 δ 和最近点对(p,q)。

算法中的几个关键点：分割线的寻找和最小距离相关的比较次数的判定

算法可以优化：比如对 p 点之后的 7 个点检查时未考虑两点均属同一侧的情况可以考虑减小比较次数。

做 DFT 时，是否总假定有 n=2m?

答：是个，总有 n=2m

什么是快速傅立叶变换（FFT）？如何用 FFT 来计算 2 个多项式的乘积？

答：能在 Θ(nlogn)时间里完成 DFT 的算法就称为 FFT.

给了 2 个多项式的系数向量 a 和 b 之后，若其系数不是 2 的幂次，则将 a 和 b 的规模扩大（向量最后加若干个 0）使得 n=2m..然后把这两个向量维数再扩大一倍，得到两个维数为 2n 的向量。分别对 2 个向量做 DFT，所得到两个向量进行点乘，再对结果做 2n 阶的 DFT-1,即可求得 2 多项式相乘后的多项式系数 c

什么是平衡？分治法与平衡有着什么样的关系？

答：在使用分治法和递归时，要尽量把问题分成规模相等，或至少是规模相近的子问题，这样才能提高算法的效率。使子问题规模尽量接近的做法，就是所谓的平衡。

分治法与动态规划法之间的相同点是什么？不同之处在哪些方面？

答：与分治法类似，动态规划法 也是把问题一层一层地分解为规模逐渐减小的同类型的子问题。

动态规划法与分治法的一个重要的不同点在于，用分治法分解后得到的子问题通常都是相互独立的，而用动态规划法分解后得到的子问题很多都是重复的。因此，对重复出现的子问题，只是在第一次遇到时才进行计算，然后把计算所得的结果保存起来；当再次遇到该子问题时，就直接引用已保存的结果，而不再重新求解。

简述求矩阵连乘最少乘法次数的动态规划算法（不超过 300 字）

答：按照做最后一次乘法的位置进行划分，

该矩阵连乘一共可分为 j-i 种情况即有(j-i)种断开方式：

Mi(Mi+1---Mj)，(MiMi+1)(Mi+2---Mj)，---，(MiMi+1---Mj-1)Mj。

其中任一对括号内的矩阵个数（即规模）不超过 j-i。

由于在此之前我们已知 任一个规模不超过 j-i 的矩阵连乘所需的最少乘法次数， 故(Mi---Mk)和(Mk+1---Mj)所需的最少乘法次数已知， 将它们分别记之为 mik 和 mk+1,j。 于是，形为(Mi…Mk)(Mk+1…Mj)的矩阵连乘所需的最少乘法次数为：mik+mk+1,j+ri-1×rk×rj。 此式中的所有参加运算的数均已知， 故此式在 O(1)时间里即可完成计算。 对满足 i≤k<j 的共 j-i 种情况逐一进行比较，我们就可以得到矩阵连乘 MiMi+1---Mj-1Mj（i<j）所需的最少乘法次数 mij 为：

mij=min（i≤k<j）{mik+mk+1,j+ri-1×rk×rj}，（*） 其计算可在 O(j-i)时间里完成。

于是在初始时我们定义 mii=0（相当于单个矩阵的情况）， 然后首先求出计算 M1M2, M2M3, ---, Mn-1Mn 所需的乘法次数 mi,i+1(i=1,2,---,n-1)， 具体数值为 ri-1×ri×ri+1， 因 mii= m i+1, i+1=0； 再利用这些结果和（*）式，就可以求出 计算 M1M2M3, M2M3 M4, ---, Mn-2Mn-1Mn 所需的最少乘法次数 mi,i+2(i=1,2,---,n-2)。 同理，可依次算出 mi,i+3(i=1,2,---,n-3)，mi,i+4(i=1,2,---,n-4)，---， 直至算出 m1,n 即 M1M2---Mn-1Mn 矩阵连乘所需的最少乘法次数。

能够用动态规划法求解的问题通常具有什么样的特征？

答：若一个问题可以分解为若干个高度重复的子问题， 且问题也具有最优子结构性质，就可以用动态规划法求解： 以递推的方式逐层计算最优值并记录必要的信息， 最后根据记录的信息构造最优解。

什么是最长公共子序列问题？在求 LCS 的算法中，C[i,j]是如何计算的？为什么需要这样计算？

答：若 Z<X，Z<Y，且不存在比 Z 更长的 X 和 Y 的公共子序列，

则称 Z 是 X 和 Y 的最长公共子序列，记为 Z∈LCS(X,Y)。如何在低于指数级的时间复杂度内找到 LCS 称为最长公共子序列问题

C[i,j]=0, 若 i=0 或 j=0

C[I,j]=C[i-1,j-1]+1 若 i,j>0 且 xi=yi

C[I,j]=max{C[i-1,j],C[i,j-1]} 若 i,j>0 且 xi!=yi

二维数组 C，用 C[i,j]记录 Xi 与 Yj 的 LCS 的长度 如果我们是自底向上进行递推计算，那么在计算 C[i,j]之前， C[i-1,j-1], C[i-1,j]与 C[i,j-1]均已计算出来。此时我们 根据 X[i]=Y[j]还是 X[i]≠Y[j]，就可以计算出 C[i,j]。

计算的理由：求 LCS(Xm-1，Y)的长度与 LCS(X，Yn-1)的长度 这两个问题不是相互独立的： ∵两者都要求 LCS(Xm-1，Yn-1)的长度，因而具有重叠

性。 另外两个序列的LCS中包含了两个序列的前缀的LCS， 故问题具有最优子结构性质 考虑用动态规划法。

用200～300字概述求最优二分搜索树算法的主要步骤。算法中有哪几点最为关键？

答：记cij是最优子树Tij的耗费， 则ci,k-1是最优子树Ti,k-1的耗费，ck,j是最优子树Tk,j的耗费。 考察以ak (i+1≤k≤j)为根、由结点bi，ai+1，bi+1，…，aj，bj构成的、 耗费最小的树的总耗费：该树的左子树必然是Ti,k-1，右子树必然是Tk,j。 这棵树的总耗费可分为三部分：左子树、右子树和根。 由于Ti,k-1作为左子树接到结点ak之下时，其耗费增加wi,k-1， 故左子树的耗费为：ci,k-1+wi,k-1， 同理，右子树的耗费为：ck,j+wk,j， 由于根ak的深度为0，按定义，根的耗费为pk。 因此，以ak 为根、耗费最小的树的总耗费为：ci,k-1+ wi,k-1+ckj+wk,j+pk。 注意到，wi,k-1=qi+pi+1+qi+1+…+pk-1+qk-1， wk,j=qk+pk+1+qk+1+…+pj+qj， 从而有wi,k-1+wkj+pk = qi+pi+1+qi+1+…+pk-1+qk-1+ pk +qk+pk+1+qk+1+…+pj+qj = wij。 由此得到以ak 为根、耗费最小的树的总耗费为：ci,k-1+ckj+wi,j

由于pi（i=1,2,…,n），qj（j=0,1,2,…,n）在初始时已经知道， 若wi,j-1已知，则根据wi,j= wi,j-1+pj + qj可以计算出wij。 故当ci,k-1与ckj已知时，以ak 为根的树的最小总耗费 在O(1)时间就可以计算出来。 分别计算以ai+1，ai+2，…，aj为根、 含有结点bi，ai+1，bi+1，…，aj，bj的树的最小总耗费， 从中选出耗费最小的树，此即最优子树Tij。 因此，最优子树Tij的耗费cij={cminj k i ≤ <j k i ≤ <i,k-1+ckj+wij}。 递推求cij及记录Tij的根的算法

本算法的关键点：分析出最优二分搜索树具有最优子结构；在计算中规模较小的最优子树在计算中要被多次用到。Cij和Wij都是可以通过前面的计算递推得出的。

有了Tij的根的序号后，如何构造出最优二分搜索树？

答：设Tij的根为ak (rij记录到的值是k)，则从根开始建结点。

```
Build-tree(i,j,r,A) /*建立最优子树Tij*/

{If i≥j return "nill";

pointer←newnode(nodetype);

k←rij; /*必有i < m ≤ j*/

pointer→value←A[k]; /*A[k]即ak*/

pointer→leftson←Buildtree(i,k-1,r,A); /*建立最优左子树Ti,k-1*/
```

pointer→rightson←Buildertree(k,j,r,A); /*建立最优右子树 Tk,j*/

return pointer;

}

Francis Yao 的办法为什么会把算法时间复杂度从 O(n3)降到 O(n2)？

答：Th: 如果最小耗费树 Ti,j-1 和 Ti+1,j 的根分别为 ap 和 aq，则必有

⑴p≤q ⑵最小耗费树 Tij 的根 ak 满足 p≤k≤q。 （证明略。）

有了上述定理，我们无需在 ai+1～aj 之间去一一尝试，使得找到的 ak 为根时，{ci,k-1+ckj+wij}为最小，而只要从 ap～aq 之间去找一个根即可。

算法时间复杂度为 Θ(n2)的证明：

首先注意 Ti,j-1 和 Ti+1,j 的规模恰好比 Ti,j 小 1。由于算法是按树中结点的个数（即规模）从小到大计算的，故在计算 rij 时，ri,j-1（即定理中的 p）和 ri+1,j（即定理中的 q）

都已经计算出来了。一般地，设含有 k 个连续结点的最优二分搜索子树的根

r0k，r1(k+1)，…，r(n-k),n 已经计算出来。由 Th 知， 含有 k+1 个连续结点的最优二分搜索子树 Ti， i+k+1 的根 必在 ri,i+k 与 ri+1,i+k+1 之间。

故 r0,k+1 在 r0,k 与 r1,k+1 之间， 求出 r0,k+1 的搜索区间长度为 r1,k+1-r0,k；

r1,k+2 在 r1,k+1 与 r2,k+2 之间，求出 r1,k+2 的搜索区间长度 r2,k+2 -r1,k+1；

r2,k+3 在 r2,k+2 与 r3,k+3 之间，求出 r2,k+3 的搜索区间长度 r3,k+3 -r2,k+2；

…… r(n-k-1),n 在 r(n-k-1),n-1 与 r(n-k),n 之间，

求出 r(n-k-1),n 的搜索区间长度为 r(n-k),n-r(n-k-1),n-1；

于是，求出所有规模为 k+1 的最优二分搜索子树的根

r0,k+1， r1,k+2,…,r(n-k-1),n 的搜索长度的总和=

(rn-k,n-rn-k-1,n-1)+…+(r3,k+3 -r2,k+2)+(r2,k+2-r1,k+1)+(r1,k+1-r0,k)=r(n-k),n-r0,k

∵rn-k,n 最多为 n，r0,k 最小为 1，

∴搜索区间长度最多为 Θ(n)，即第 2 层循环执行总次数为 Θ(n)，

因此算法总的时间复杂度为 Θ(n2)。

用 200～300 字概述二维流水作业调度算法的主要步骤

答：求解该问题的算法如下。

1. 建立长为 2n 的数组 C，将 a1, a2,---, an 依次放入 C[1]~ C[n]中，b1, b2,---, bn 依次放入 C[n+1]~ C[2n]中。 /* O(n)， 下面将对这 2n 个数进行排序*/

2. 对长为 2n 的数组 D 进行初始化：D[1]~ D[n]中依次放 1,2,---,n,D[n+1]~ D[2n]中依次放-1,-2,---,-n。/*O(n)， 分别对应于 a1, a2,---, an 和 b1, b2,---, bn 的下标*/

3. 对数组 C 进行排序，D[k]始终保持与 C[k]的对应关系。

（若 C[i]与 C[j]对换，则 D[i]也与 D[j]对换。

或将 C，D 放在同一结构体中。） 当 a1, a2,---, an 及 b1, b2,---, bn 按从小到大次序排好之后，

D[1]~ D[2n]也就按从小到大的次序记录了这些 ai 和 bj 的下标即作业号(bj 的下标前有一负号以区别于 ai））。/*O(n log n)*/

4. 将 E[1]~ E[n]全部置为"No"。 /* O(n)，表示所有任务均尚未被安排*/

5. 下标变量初始化：i←1； j←n； k←1；

/*O(1)，i 指向当前最左空位 F[i]，放当前应最先安排的作业号;*/ /* j 指向当前最右空位 F[j]，放当前应最后安排的作业号;*/ /* k 从 1 开始逐次增 1，*/ /* D[k]（或-D[k]）按 ai 和 bj 从小到大的次序依次给出作业号*/

6. while i ≤ j

do { /* 作业尚未安排完毕，i 从小到大，j 从大到小*/

if D[k] > 0 then {if E[D[k]]为"No"then /*作业 D[k]放在当前最左空位*/ {F[i]←D[k]; i 增 1; E[D[k]] 置为"Yes"}}

else if E[-D[k]]为"No"then /*作业-D[k]放在当前最右空位*/ {F[j]←-D[k]; j 减 1; E[-D[k]] 置为"Yes"}}

k 增 1；}

什么是备忘录方法？它在什么情况下使用较为有效？

答：若有大量的子问题无需求解时，用备忘录方法较省时。

但当无需计算的子问题只有少部分或全部都要计算时，

用递推方法比备忘录方法要好（如矩阵连乘，最优二分搜索树）。

简单不相交集的合并算法中为什么要引进集合的外部名和内部名？

答：：若没有内部名，

则每次合并时两个集合中的所有元素均要改名（改为 K），

这样，在 n-1 次 Union 中改名的时间就变为 O(n2) 。

什么是平摊分析？平摊分析常用的手法有哪几种？简单说明这几种手法的要点。

答：考虑 n 条指令执行的最坏时间复杂性。 即使某些指令执行时具有比较大的代价，

但利用平摊分析后对整体考虑， 可以得到较小的平均代价。

平摊分析方法主要有：聚集方法，会计方法和势能方法。

聚集方法：全局考虑时间复杂性， 把 n 条指令的耗费分为几类； 分别计算每一类耗费的总和， 然后再把各类耗费总加起来。

会计方法：利用几个操作之间的联系，在一个操作中预先支付下面某个操作的费用，以达到简化代价计算的目地。

势能方法：设 Ci 为第 i 个操作的实际代价，

D0 为处理对象的数据结构的初始状态，

Di 为第 i 个操作施加于数据结构 Di-1 之上后数据结构的状态，

引入势函数 Φ， Φ(Di)是与 Di 相关的势。

定义第 i 个操作的平摊代价为：

= Ci +(Φ(Di)- Φ(Di-1))（即实际代价加上势的变化），

为什么树结构下执行 O(n)条带路径压缩的 Union-Find 指令只需要 O(n*G(n))时间？

答：用平摊分析的聚集方法，把 O(n)条 Find 指令的耗费分为三类：

1） O(n)条 Find 指令的根费用，

2） O(n)条 Find 指令的组费用，

3） O(n)条 Find 指令的路径费用。

根费用：执行一条 Find 指令时，处理根及其儿子所需的费用。 一条 Find 指令只会碰到一个根（及其儿子）， 故 O(n)条 Find 指令的根费用为 O(n)， 这样，根及其儿子的费用已全部计算在内。

组费用：若结点 ik (0≤k≤m-2)与其父结点 ik+1 的秩不在同一个秩组中，则对 ik 收取一个组费用。因最多有 G(n)个组， 故一条 Find 指令最多只会碰到 G(n)个结点、 其秩与其父结点的秩不在同一个秩组中， 故 O(n)条 Find 指令的组费用最多为 O(n*G(n))。

路径费用：由于其秩在组号为 g 的组中结点的个数不超过 n/F (g)， 故组 g 中的结点的收取的路径费用 不超过[n/F(g)]*[F(g)-(F(g-1)+1)]< [n/F(g)]*F(g)=n。 因总共只有 G(n)个组， 故所有结点在 O(n)条 Find 指令的执行中， 收取的路径费用不超过 O(n*G(n))。

三项费用总加起来，有 O(n)+ O(n*G(n))+O(n*G(n))=O(n*G(n)) 于是可得结论：如果合并是是把小集合并入大集合， 且执行 Find 指令时实施路径压缩，则执行 O(n)条 Unoin-Find 指令的时间复杂度为 O(n*G(n))；

树结构下执行 O(n)条带路径压缩的 Union-Find 指令能否降到线性即 O(n)时间？

答：执行 O(n)条 Unoin-Find 指令时，对于任意的 c，都存在一个特殊的 Unoin-Find 指令序列，

使得执行该序列的时间复杂度>cn， 即算法在最坏情况下不是线性的。

用 200～300 字概述 Link(v,r)程序的执行过程。该程序的要点在什么地方？

答：要点在于：为保持 Weight 的性质，合并的同时要对 Weight[r']进行修改。（由于原先的 Link 就是把 r 接到 v 之下，而含有 v 的树中各结点的深度并未受到任何影响，现在把 Tr 接在 v'之下，Tv 中 的各结点当然也不会受到任何影响， 故 Tv 树中各结点的 Weight 值无需修改。）Tr 树中各结点的 Weight 值如何修改？

设用 Find-Depth 指令可以查到 v 的深度为 Depth(v)，

你认为脱线 MIN 算法的关键点在什么地方？

答：算法：

```
for i=1 to n do {
j←FIND(i);
if j=0 then {输出"i 未在序列中出现"}
else if j>k then {输出"i 未被删除"}
else{ /* i 确实被删除了*/
输出"i 被第 j 条 E 指令所删除";
UNION(j, Succ[j], Succ[j]);
Succ[Pred[j]]←Succ[j]; /* 集合 j 不再存在*/
Pred[Succ[j]]←Pred[j]
} }
```

算法最关键的在于集合的 Union 和 find 算法的使用。

比较 Las Vegas 算法和 Monte Carlo 算法，它们有什么相同和相异之处？随机算法有哪些优点？答：LasVegas 算法总是给出正确的结果，

但在少数应用中，可能出现求不出解的情况。此时需再次调用算 法进行计算，直到获得解为止。对于此类算法，主要是分析算法的时间复杂度的期望值，以及调用一次产生失败（求不出解）的概率。Mont Carlo 算法通常不能保证计算出来的结果总是正确的，一般只能断定所给解的正确性不小于 p（＜p＜1＝。通过算法的反复执行（即以增大算法的执行时间为 代价），能够使发生错误的概率小到可以忽略的程度。由于每次执行的算法是独立的，故 k 次执行均发生错误的概率为(1-p)k。

随机算法的优点：

1. 对于某一给定的问题，随机算法所需的时间与空间复杂性， 往往比当前已知的、最好的确定性算法要好。

2. 到目前为止设计出来的各种随机算法，无论是从理解上还是实现上，都是极为简单的。

3. 随机算法避免了去构造最坏情况的例子。 最坏情况虽有可能发生，但是它不依赖于某种固定的模式，只是由于"运气不好"才出现此种情况。

简述随机取数算法和找第 k 小元素的随机算法。

答：Random Sampling 问题（Las Vegas 算法）

设给定 n 个元素（为简单起见，设为 1，2，…n），要求从 n 个数中随机地选取 m 个数（m≤n）。可以用一个长为 n 的布尔数组 B 来标识 i 是否被选中，初始时均表为"未选中"。然后随机产生〔1，n〕之间的一个整数 i，若 B[i]为"未选中"，则将 i 加入被选中队列， 同时把 B[i]标识为"已选中"，反复执行直到 m 个不同的数全部被选出为止。

找第 k 小元素的随机算法（Las Vegas 算法）：

在 n 个数中随机的找一个数 A[i]=x，然后将其余 n-1 个数与 x 比较，分别放入三个数组中： S1(元素均<x)，S2(元素均=x)，S3(元素均＞x)。 若|S1|≥k 则调用 Select(k,S1)； 若（|S1|+|S2|）≥k，则第 k 小元素就是 x； 否则就有（|S1|+|S2|）＜ k，此时调用 Select(k-|S1|-|S2|,S3)。

什么是 Sherwood 随机化方法？简述 Testing String Equality 算法的误判率分析。

答：Sherwood 随机化方法（属 Las Vegas 算法） 如果某个问题已经有了一个平均性质较好的确定性算法， 但是该算法在最坏情况下效率不高，此时引入一个随机数发生器 （通常是服从均匀分布，根据问题需要也可以产生其他的分布），将一个确定性算法改成一个随机算法，使得对于任何输入实例， 该算法在概率意义下都有很好的性能（e.g. Select, Quicksort 等）。 如果算法（所给的确定性算法)无法直接使用 Sherwood 方法，则可以采用随机预处理的方法，使得输入对象服从均匀分布 （或其他分布），然后再用确定性算法对其进行处理。 所得效果在概率意义下与 Sherwood 型算法相同。

数论定理 1：设 π(n)是小于 n 的素数个数，则 π(n)≈nnelog， 误差率不超过 6%。

∵M=2n2，Pr[failure]的分母 π(M)≈

Pr[failure]的分子 ≤ 使得 Ip(x)=Ip(y)的素数 p(p＜M)的个数 =能够整除｜I(x)-I(y)｜的素数 p(p＜M)的个数 （∵a≡b (mod p) iff p 整除｜a-b｜）

数论定理 2：如果 a＜2n，则能够整除 a 的素数个数不超过 π(n)个。 （只要 n 不是太小。）

∴Pr[failure]≤π(n) / π(M)≈nnnneelog/log/2=n1。即误匹配的概率小于 n1。

设 x≠y，如果取 k 个不同的小于 2n2 的素数来求 Ip(x)和 Ip(y)， 由于事件的独立性,因此事件均发生的概率满足乘法规则，即 k 次试验均有 Ip(x)=Ip(y)但 x≠y（误匹配）的概率小于 kn1。 ∴当 n 较大、且重复了 k 次试验时，误匹配的概率趋于 0。

计算模型 RAM 与 RASP 之间有什么相同和相异之处？

答:RAM 和 RASP 的相同之处在于作为计算模型都有各种寻址指令，且任何一个 RAM 程序，都可由一个 RASP 程序来模拟实现， 且时间复杂性数量级相同（不论哪一种耗费标准）。

即如果 RAM 程序的时间复杂度为 T(n)，则 RASP 实现同 样功能的程序的时间复杂度为 kT(n)（k 为常数）。任何一个 RASP 程序都可由 RAM 程序来模拟， 且时间复杂性的量级相同（无论哪一种耗费标准）。RASP 程序与 RAM 程序之间可以相互模拟， 且无论哪一种耗费标准，两者的时间复杂性的量级均相同。

不同之处在于：RAM 的程序一经给定就不允许修改；而 RASP（Random Access Stored Program）的程序可以修改，只要将程序放在存储器上就可以。RASP 程序没有间接寻址

Alan Turing 是怎样对人类计算过程进行概括的？

答：Turing 根据这个过程构造出了一个计算模型，称之为 Turing 机。

这个计算模型有一条带子（带子相当于一张纸），带子上有无穷多个方格，每个方格中可以写给定有穷符号表中的一个字符。有一个读写头（读相当于用眼睛来看，写相当于擦或写）。 还有一个有穷状态控制器 FSC（相当于人脑），FSC 根据当前读到的符号（相当于眼睛看到的）和

自己当前的状态（相当于人脑的决策）决定三件事：

改变读写头当前所指的字母（相当于在纸上改写一些符号），改变（或不改变）自己当前的状态（准备下一步的计算），使读写头左移或右移或不动（相当于改变眼睛看到的范围）。

Church-Turing Thesis 的内容是什么？它有什么意义？

答：任何合理的计算模型都是相互等价的（计算范围相同）。

合理：单位时间内可以完成的工作量，有一个多项式的上限。不合理举例：任意多道的并行计算。由于有"任何"二字，故无法进行证明。但迄今为止所有被提出的合理计算模型均满足该论题。

这个论题说明：可计算性本身是不依赖于具体模型的客观存在。

RAM 程序、RASP 程序与 Turing 机相互之间的时间复杂度关系是怎样的（考虑两种耗费标准）？

答：任何一个 RAM 程序，都可由一个 RASP 程序来模拟实现，且时间复杂性数量级相同（不论哪一种耗费标准）。即如果 RAM 程序的时间复杂度为 T(n)，则 RASP 实现同样功能的程序的时间复杂度为 kT(n)（k 为常数）。

当一台 TM 的时间复杂度为 T(n)时，模拟该 TM 的 RAM 程序的对数耗费不超过 O(T(n)logT(n))；而当一不含乘、除法指令的 RAM 程序的对数耗费为 T(n)时，模拟该程序的 TM 的时间复杂度不超过 O(T2(n))；当一含有乘、除法指令的 RAM 程序的 对数耗费为 T(n)时，

模拟该程序的 TM 的时间复杂度不超过 O(T3(n))。

在对数耗费的标准下， RAM，RASP 与 TM 是多项式相关的计算模型。

RASP 程序如何模拟 RAM 程序？RAM 如何模拟 Turing 机？ Turing 机如何模拟 RAM?

答：RASP 程序按下图模拟 RAM 程序：

在模拟过程中，RASP 的第 r+i 号寄存器 始终对应于 RAM 的第 i 号寄存器，RASP 的 R1 作为暂存器。 R0 与 r0 在一条指令的模拟过程中可能不对应，但在一条指令模拟的开始和结束时一定对应。

模拟非间址寻址指令时，先将指令对应的编码放入 j 寄存器，若 操作数为"=i"，则去掉"="，将 i 放入 j+1 寄存器，若操作数为"i"，要看 i 是否为 0：i 为 0 时，将 0 放入 j+1 寄存器，i>0 时，将 r+i 放入 j+1 寄存器。间址寻址的 RAM 指令，可以用一组 RASP 指令来模拟实现。

RAM 模拟 TM 的方法如下图：

模拟时存储内容的对应方法按上图所示。TM 的 k 个带头的位置 分别放在 RAM 的单元 r1 至 rk 中，即第 j 个单元的内容 C(j)就是第 j 个带头在第 j 条带上的当前位置。c 为 RAM 存储器中存放 a0 的单元之前的单元编 号。k 条带上的首格内容（分别为 a0,b0,…,z0）存放在 第 c 号单元后的 k 个单元中；其后再放 k 条带上的次格内容（分别为 a1,b1,…,z1）；因此，TM 的第 j 条带当前扫描着的单元的内容是存放在 RAM 存储 器的第 c+k*C(j)+j 个单元中，即内容为 C(c+k*C(j)+j)。

（C(j)是第 j 个带头的当前位置，而位置的计数是从 0 开始的）。e.g.：第二条带第 1 个格子中的内容放在第 c+k*1+2 号单元。

TM 模拟 RAM 的方法如下图：

用 5 带 TM 来模拟 RAM 程序的过程：

带 1 用来记录内容为非 0 的 RAM 寄存器的编号及该寄存器的内容。带 2 存放 RAM 累加器 r0 中的内容 α0，带 3 作为暂存工作带。带 4 上放 RAM 输入带上的内容，带 5 上放输出。

Turing 机所接受的语言是什么样的语言？各种语言之间有什么样的关系？

答：Turing 可计算函数部分递归函数。

递归可枚举集>（完全）递归集>原始递归集

部分递归函数>完全递归函数>原始递归函数

非确定性 Turing 机与确定性 Turing 机的主要区别在什么地方？

答：与 DTM 不同的是，NDTM 的每一步动作允许有若干个选择，对于给定的 Q×Tk 的一个元素（qi，a1，a2,…，ak），它的 δ 转移函数值不是对应于一个 Q×(T×{L,R,S})k 中的一个元素，而是对应于 Q×(T×{L,R,S})k 中的一个子集。

与 DTM 不同的是，DTM 的 ID 序列是线性的：ID0 ⊢ ID1 ⊢ ID2 ⊢ … ⊢ IDm，而 NDTM 的 ID 序列通常是用树来描述的 （因为在每个格局都可能有多个选择）。

NDTM 的另一种解释是，每当遇到 n 个（n≥2）选择时， NDTM 就把自身复制 n 个，让它们进行并行的计算。 由于具有任意多道并行计算的能力，

非确定性 Turing 机的时间复杂度是如何定义的？

答：NDTM 的时间复杂度 T(n)：对于可接受的输入串， T(n)=max{关于输入 ω 的时间复杂度 ｜ ω 的长为 n 且被该 NDTM 接受} 即先求出每一个被接受 ω 的最短路径长度，然后对这些长度求最大。

什么是 P 类与 NP 类语言？如不用非确定性 Turing 机的概念，如何定义 P 类与 NP 类的问题？答：语言族 P=｛L|L 是一个能在多项式时间内被一台 DTM 接受的语言｝

NP=｛L|L 是一个能在多项式时间内被一台 NDTM 接受的语言｝

不需要 NDTM 概念的 NP 问题的定义

设 S 是一个集合，若某个判定问题 A 的所有解均属于 S， 则 S 中的一个元素称为问题 A 的一个证书（certificate），S 被称为是 A 的证书所构成的集合。对判定问题类 A,若存在一个由 A 的证书所构成的集合 S（S 中含有 A 的全部解），且存在一个算法 F，对 S 中的每一个证书 α， F 可在多项式时间里验证 α 是否为 A 的一个解，则称 A∈NP。 （该定义与用 NDTM 概念的 NP 类定义是等价的。）

称 P 类问题是多项式时间可解的。 称 NP 类问题是多项式时间可验证的。

Karp 多项式规约是如何表述的？它与 Cook 多项式规约之间的关系如何？

答：Karp 规约：设 L、L0 分别是字母表Σ、Σ0 上的语言（字符串的集合），

若存在两个闭包间的一个变换 f：Σ*→Σ*，满足

1） ∀ω∈Σ*，ω∈L⇔f(ω)∈L0；

2） 若 ω 的长为 n，则 f(ω)的长不超过 PL(n)，这里 PL(n)是一个多项式；

3） 任取 ω∈Σ*，设 ω 的长度为 n，则 f （ω)在多项式时间 p(n)内可计算。

则称 L 可多项式变换为 L0，记作 L≤p L0。

它和 Cook 规约的关系是 Karp 的要求严，Cook 的要求松，即满足 Karp 就满足 Cook。

至今尚未发现满足 Cook 而不满足 Karp 的语言，即两者大体等价。Karp 的定义简洁，Cook 的定义繁琐。

什么是 NP-完全性语言？什么是 NP-完全性问题？什么是 NP-hard 问题？

答：NP-完全性语言

定义 1（狭义，Karp）：称满足下述 2 条的语言 L0 是 NP-C 的：

1)L0∈NP; 2) ∀L∈NP，都有 L≤pL0。

NP-完全性问题 ：若某个判定问题进行编码后，所对应的语言 L0 是 NP-C 的， 则称该问题是 NP-C 的。

有些最优化问题（对应的编码 ω∈L0）可以满足 NP-完全性定义的第 2 条要求：∀L∈NP，都有 L≤p L0。 满足上述条件的问题被称为 NP-hard 问题。

引入 NP-完全性概念有什么意义？列举 20 个 NP-完全性或 NP-hard 问题。

答：如果存在一台 DTM 在多项式时间里接受某个 NP-C 语言，

则所有 NP 类语言均可找到 DTM 在多项式时间里接受，从而有 P=NP。

如果某个 NP 类语言不存在 DTM 在多项式时间里接受（即 P≠NP），

则所有 NP-C 语言都不存在 DTM 在多项式时间里接受，

即有 NP-C∩P=Φ。

什么是算法 A 对于实例 I 的近似比、A 的绝对近似比和渐进近似比？

答：算法 A 对于实例 I 的近似比(ratio factor)
=RA(I)=max{A(I)/OPT(I),OPT(I)/A(I)}

由于是取 max，故近似比总是大于等于 1 的。

绝对近似比 rA=Inf{RA(I)} 对一切 即算法 A 对于一切实例 I 的近似比的最小上界（相当于最大值）。

渐进近似比 rA=sup{r≥1| 存在正整数 n0，使得对于所有 OPT(I)≥n0 的实例，都有≤r} 即反映了近似比的收敛情况，它允许 OPT(I)较小的实例的近似比大于 rA。

什么是多项式时间近似方案（PTAS）？什么是完全多项式时间近似方案（FPTAS，FPAS）？

答：多项式时间近似方案 （PTAS, Polynomial Time Approximation Scheme）

设 ε 是求解某类问题的多项式时间近似算法，

ε >0 也作为该算法的输入。如果对每一个给定的 ε，

的绝对近似比≤1＋ε，则称 A 是一个 PTAS。

完全多项式时间近似方案 （FPTAS，FPAS, Fully Polynomial Time Approximation Scheme）

如果一个 PTAS 以某个二元多项式函数 p(|I|,1/ε)

为时间复杂度上界，则称 ε 是一个 FPTAS。

什么是伪多项式时间的算法？如何用动态规划法求解背包问题？

答：定义：设实例 I 的输入规模为 n，实例中的最大数为 max(I)， 若算法的时间复杂性以某个二元多项式 p(n,max(I))为上界， 则称该算法是伪多项式时间的算法。

动态规划法。对 0≤k≤n，0≤b≤B，设 f(b)为：

装前 k 件物品中若干件、且体积和不超过 b 时可得到的最大价值。

因此，f(B)就是该问题的最优解。

根据 f(b)的定义可知子问题具有最优子结构性质。

另外，引入初值均为 0 的二维数组 x，每一个数组元素 x[k,b]记录：

当体积限制为 b 时，uk 是否被选中，1 为被选中，0 为未被选中。

初始当 k=0 时，有(b)=0（0≤b≤B），此为递推计算的基础值。

设对 b∈[0,B]，(b)，(b)，…，f(b)均已计算出来，

此时需要对区间[0,B]中的整数 b（即 0≤b≤B），逐一求出(b)。

考虑 b 与 sk 的关系。

若 b ＜ sk，则对于体积限制 b，物品 uk 根本不能装（太大了），

∴x[k,b]应为 0（即此时不选物品 uk）；同时执行 fk(b)←fk-1(b)。

（实际执行时，只要对小于 sk 的 b，执行 fk (b)←fk-1 (b)。）

若 b≥sk，就要看把一些物品取出，再把物品 uk 放入，

所得的总价值是否比不装物品 uk（其总价值为 fk-1 (b)）时大。

而装物品 uk 时，可获得的最大价值为 fk-1 (b-sk)+ck。

这里的 fk-1 (b-sk)是：当体积限制不超过 b-sk 时，

装前 k-1 件物品中的若干件，可得到的最大价值。

若 fk-1 (b-sk)+ck ＞(b)，则物品 uk 应装入，

即此时要执行 x[k,b]←1，以及 fk (b)←fk-1 (b-sk)+ck。

否则应执行 x[k,b]←0，以及 fk (b)←fk-1 (b)。

按上述方法，当执行到 k=n，b=B 时，最优解的值即为(B)。

这部分算法的时间复杂度为 O(nB)。

上述计算完毕之后，再根据 x 数组内容，找出应取哪几件物品。

置初值 b=B，循环从 j=n 开始，检查 x[j,b]：

若 x[j,b]=0，则表示 uj 未被选中，故不做动作，直接执行 j←j-1；

若 x[j,b]=1，则表示 uj 被选中，把 j 放入集合 S 中（S 初始为空），

然后执行 b←b-sj 以及 j←j-1。

循环一直执行到 j=1 的判断完成后为止。算法时间复杂度为 O(n)。

从而整个算法的时间复杂度为 O(nB)。

注意 B 与输入规模 n 无关，故 nB 不是输入规模的多项式函数。

因此，算法是伪多项式时间的算法。

NP 的假定之下，可以分成哪 4 类（举例）？≠NP-hard 问题在 P

答：NP-hard 问题在 P≠NP 的假定之下，可以分成 4 类：

1、 有 FPTAS（e.g.Knapsack）

2、 有 PTAS 而没有 FPTAS（e.g. k-Knapsack）

3、 没有 PTAS，但有绝对近似比为常数的近似算法(Bin-Packing)

4、 没有绝对近似比为常数的近似算法(e.g. TSP)

为什么当 k≥2，k 背包问题不存在 FPTAS，除非 P=NP?

答：反证法：设 A 是该问题的 FPTAS，时间复杂度上界为 P(|I|,1/ε)，

对每个实例 I，令 ε=1/2ncmax，将 I 和 ε 作为 A 的输入，

由于 A 是 FPTAS，故有 OPT(I)≤(1+ε)A(I)，

∴0≤OPT(I)-A(I)≤εA(I)≤ε≤εncmax=。

但 OPT(I)和 A(I)都是正整数，∴OPT(I)=A(I)，

故 A(I)是最优化算法，其时间复杂度上界为 p(n，2ncmax)。

利用算法 A 来构造 k 等分割问题的算法 A'：

对于 k 等分割的任一实例 I，即 n 个正整数 sj (1≤j≤n)，

构造 k 背包问题的实例 I'：物品 j 的体积为 sj，

价值 cj=1(1≤j≤n)，背包容量 Bi=(1≤i≤k)。

将该实例 I'和 ε==作为 A 的输入，求得 I'的最优解 A(I')；

当且仅当 OPT(I')=n（把所有物品均装入 k 个背包）时，

A'对 I 回答为"Yes"。

注意，k 等分割有解 iff A'回答为"Yes"

由于根据 I 构造 I'仅需 O(n)时间，|I'|=|I|=n， ε =，

而 A 算法的时间复杂度上界不超过 p(n,)=p(n,2n)，

是关于 n 的多项式，∴A'是 k 等分割问题的多项式时间算法，

即找到了一个求解 NP-C 问题的多项式时间算法，

∴P=NP，与 P≠NP 矛盾。

结论：假设 P≠NP，则 k 背包问题没有 FPTAS。

2014 算法设计与分析课程试题题库含答案

1、 一个算法应有哪些主要特征？

又穷性，确定性，可行性，0 个或多个输入，一个或多个输出

2、 分治法（Divide and Conquer）与动态规划（Dynamic Programming）有什么不同？

分治算法会重复的求解公共子问题，会做许多不必要的工作，而动态规划对每个子问题之求解一次，将其结果存入一张表中，从而避免了每次遇到各个子问题有从新求解。

3、 试举例说明贪心算法对有的问题是有效的，而对一些问题是无效的。

贪心算有效性：最小生成树、哈弗曼、活动安排、单元最短路径。

无效反例：0——1 背包问题，无向图找最短路径问题。

4、 求解方程 f(n)=f(n-1)+f(n-2)，f(1)=f(2)=1。

由 f(n)=f(n-1)+f(n-2)可得

f(n)-f(n-1)-f(n-2)=0,可得方程的特征方程为

$x^2-x-1=0$,设特征方程的2个根本分别为x_1,x_2,则可得

$x_1=x_2=\frac{1+\sqrt{5}}{2}$,则有

$f(n)=c_1(\frac{1+\sqrt{5}}{2})^n+c_2(\frac{1-\sqrt{5}}{2})^n$

又$f(1)=f(2)=1$可得

$$\begin{cases} f(1)=\frac{1+\sqrt{5}}{2}c_1+\frac{1-\sqrt{5}}{2}c_2=1 \\ f(2)=(\frac{1+\sqrt{5}}{2})^2c_1+(\frac{1-\sqrt{5}}{2})^2c_2=1 \end{cases}$$

可得$c_1=a,c_2=b$

$f(n)=a(\frac{1+\sqrt{5}}{2})^n+b(\frac{1-\sqrt{5}}{2})^n$

5、 求解方程 T(n)=2T(n/2)+1，T(1)=1，设 $n=2^k$。

$T(n)=2T(n/2)+1$

$2T(n/2)=2^2T(n/2^2)+2$

$2^2T(n/2^2)=2^3T(n/2^3)+2^2$

.

.

.

$2^{k-1}T(n/2^{k-1})=2^kT(n/2^k)+2^{k-1}$

上面所有式子相加，相消得

$T(n)=2^kT(1)+2^0+2^1+2^2+\ldots\ldots+2^{k-1}$

$=2^k+1*\frac{1-2^k}{1-2}$

$=2^{k+1}-1$

6、 编写一个 Quick Sorting 算法，并分析时间复杂性。

```
int part(int *a,int p,int r){
    int i,j,x,t;
    x=a[r];
    i=p-1;
    for(j=p;j<=r-1;j++){
     if(a[j]<=x){
          i++;
          t=a[i];
          a[i]=a[j];
```

```
            a[j]=t;
        }
        }
        t=a[i+1];
        a[i+1]=a[r];
        a[r]=t;
        return i+1;
}
void quicksort(int *a,int p,int r){
        int q;
        if(p<r){
         q=part(a,p,r);
         quicksort(a,p,q-1);
         quicksort(a,q+1,r);
        }
}
```

快速排序时间复杂度最坏情况为 $O(n^2)$,平均为 O(nlogn);

7、 利用 Quick Sorting 的原理，编写一个查找第 k 小元素的算法。

K 不能大于数的长度。

```
int part(int *a,int p,int r){
        int i,j,x,t;
        x=a[r];
        i=p-1;
        for(j=p;j<=r-1;j++){
            if(a[j]<=x){
                i++;
                t=a[i];
                a[i]=a[j];
                a[j]=t;
            }
        }
        t=a[i+1];
        a[i+1]=a[r];
        a[r]=t;
```

```
    return i+1;
}
int k_last(int *a,int i,int j,int k){
     int mid;
     mid=part(a,i,j);
     if((mid+1)==k){
        printf("Right\n");
        return a[mid];
     }
    else if((mid+1)>k)return k_last(a,i,mid-1,k);
    else if((mid+1)<k)return k_last(a,mid+1,j,k);
}
```

8、 编写一个 Heap Sorting 算法，并分析时间复杂性。

```
void max_heapify(int *A,int i){                          //保持最大堆性质
    int largest,t,l,r;
    l=2*i;r=2*i+1;
    if(l<=heap_sizeA&&A[l]>A[i]) largest=l;
    else largest=i;
    if(r<=heap_sizeA&&A[r]>A[largest]) largest=r;
    if(largest!=i){
        t=A[i];
        A[i]=A[largest];
        A[largest]=t;
        max_heapify(A,largest);
    }
}
void build_max_heap(int *A,int lenth){          //建堆
    int i;
    heap_sizeA=lenth;
    for(i=lenth/2;i>=1;i--){
        max_heapify(A,i);
    }
}
heapsort(int *A,int lenth){
    int i,t;
    build_max_heap(A,lenth);
    for(i=lenth;i>=2;i--){
        t=A[1];
```

```
            A[1]=A[i];
            A[i]=t;
            heap_sizeA--;
            max_heapify(A,1);
        }
    }
```

9、 编写一个算法，可以检测一个字符串是否回文（如：afaddafa，abwba 等）。

```
int fun(char *A,int n){
    int i ,j;
    i=0,j=n-1;
    while(i<=j){
        if(A[i]!=A[j])break;
        i++;j--;
    }
    if(i<=j)return 0;
   else return 1;
}
```

10、 如果是检测一个字符串中是否包含一个回文子串，应如何编写算法。如果是找最大的回文子串呢？

```
int judge(char *A,int n0, int n1){
    int i,j;
    i=n0;j=n1;
    while(i<=j){
        if(A[i]!=A[j])break;
        i++;j--;
    }
    if(i<=j)return 0;
   else return 1;
}
int fun(char *A,int n){
    int i,j,k;
    for(i=1,i<=n-1;i--){
        for(j=0;j<n-i;j++){
            if(judge(&A,j,j+i)==1){output(&A,j,j+i);return 1;}
        }
    }
    return 0;
}
```

如果找最大回文则需更改为；

```
int fun(char *A,int n){
```

```
    int i,j,k;
    for(i=n-1;i>=1;i--){
        for(j=0;j<=n-i-1;j++){
            if(fun(&A,j,i+j)==1){output(&A,j,i+j);return 1;}
        }
    }
    return 0;
}
```

此时最先找到的即为最大回文。

11、 设n个不同的整数排好序后存在数组T[1：n]中。若存在一个下标i，使得T[i]=i，设计一个有效的算法找到该下标。要求时间复杂性是O(logn)。

```
void fun(){
    int low=1,high=n,mid;
    mid=(low+high)/2;
    while(low<=high){
        if(T[mid]==mid)break;
        else if(T[mid]>mid)high=mid-1;
        else low=high+1;
        mid=(low+high)/2;
    }
    if(low<=high)printf("%d",mid);
}
```

12、 编写一个采用KMP的算法查找T的子串P，并判断匹配失败的字符是否在P中，以此确定可滑动的最大距离。

```
T[1……lenthT]，P[1……lenthP]；
int next[100];
int lenthT,lenthP;
void get_next(char *P){
    int i,j;
    i=1;j=0;
    next[1]=0;
```

```
    while(i<=lenthP){
        if(j==0||P[i]==P[j]){
            ++i;++j;
            if(P[i]!=P[j]) next[i]=j;
            else next[i]=next[j];
        }
        else j=next[j];
    }
}
int KMP(char *T,char *P){
    int i,j;
    i=1;j=1;
    get_next(P);
    while(i<=lenthT&&j<=lenthP){
        if(j==0||T[i]==P[j]){++i;++j;}
        else j=next[j];
    }
    if(j>lenthP)return i-lenthP;
    else return 0;
}
```

13、 考虑一个“模糊”的算法查找 T 的子串 P，先快速找到与 P 相似的子串，再进一步确认之。

```
int new_get_index(char *T,char *P)
{
    int a,b,c;
    a=0;b=strlen(p);c=(a+b)/2;
    int k=0;
    while(){
        if(P[a]!=T[k+a]||P[b]!=T[k+b]||P[c]!=T[k+c]){k++;continue;}
        for(int j=0;j<b;j++)
```

```
            if(P[j]!=T[j+k])break;
        }
        if(j==b)return k;
    }
```

14、 设计一个算法确定 K 个矩阵相乘的最优次序，并分析该算法的时间复杂性。

```
int m[100][100];
int s[100][100];
void fun(int *p,int n){
    int i,j,k,l,q;
    for(i=1;i<=n;i++) m[i][i]=0;
    for(l=2;l<=n;l++){
        for(i=1;i<=n-l+1;i++){
            j=i+l-1;
            m[i][j]=32767;
            for(k=i;k<=j-1;k++){
                q=m[i][k]+m[k+1][j]+p[i-1]*p[k]*p[j];
                if(q<m[i][j]){
                    m[i][j]=q;   s[i][j]=k;
                }
            }
        }
    }
}
```

时间复杂度为 $\Theta(n^2)=\binom{n}{2}+n$

15、 设计一个算法从数 A[1：n]中同时找出最大元素和最小元素，只需要不超过 1．5n－2 次比较。

```
#include "stdafx.h"
#include"stdio.h"
#define N 100
int b[N];
int l=0;
int Max(int *a,int m,int n){                    //求最大值
    int x1,x2;
    if(n-m<=1){
        if(n-m==0){ b[l]=a[m];l++;return m;}          //将底层比较的较小值
```

```
存入 b[n]
            if(a[n]>=a[m]){b[l]=a[m];l++;return n;}
            else{b[l]=a[n];l++;return m;}
        }
        else {
            x1=Max(a,m,(m+n)/2);
            x2=Max(a,(m+n)/2+1,n);
            if(a[x1]>=a[x2])return x1;
            else return x2;
        }
}
int Min(int *b,int m,int n){                                    //求最小值
             int x1,x2;
             if(n-m<=1){
                 if(m==n)return m;
                 else if(b[n]<b[m])return n;
                 else return m;
             }
             else{
                 x1=Min(b,m,(m+n)/2);
                 x2=Min(b,(m+n)/2+1,n);
                 if(b[x1]<=b[x2])return x1;
                 else return x2;
              }
}

void main(){
        int a[N];
        int s,i,n;
        printf("Please input the number :");
```

```
    scanf("%d",&n);
    printf("Please input :");
    for(i=0;i<n;i++){
        scanf("%d",&a[i]);
    }
    s=Max(a,0,n-1);
    printf("Max: %d\n",a[s]);
        s=Min(b,0,l-1);
     printf("Min: %d\n",b[s]);

}
```

16、用分治法一个算法从数 A[1：n]中同时找出最大元素，分析算法的时间复杂性，并思考为什么是这样的结果。

大家自己看看，呵呵！

17、 在一个数组中出现次数最多的元素称为“众数”，编写一个找出众数的算法，并分析算法时间复杂性。

```
int most(int a[],int length)   //length 数组长度
{
    int i,j,k;
    int max,value;
    int num[100][100];
    num[0][0]=a[0];
    num[0][1]=1;
    j=0;
    for(i=1;i<length;i++)
    {
       for(k=0;k<j;k++)
       {
           if(a[i]==num[k][0])
```

```
            {
                num[k][1]++;
                break;
            }
        }
        if(k>j)
        {
            j++;
            num[j][0]=a[i];
            num[j][1]=1;
        }
    }
    max=0;
    for(k=;k<j;k++)
    {
      if(num[k][1]>max)
      {
        max=num[k][1];
        value=num[k][0];
      }
    }
    return value;                    //出现次数最多的数字
}
```

18、 设计一个使用 Hash 法的算法在数组中找出“众数”，并分析时间复杂性。

```
Hash(long T[],int n){
    struct node *temp,*p;
    struct node *array[1000];
    int i,temp,num;
    int max=1;
    num=sqrt(n);
```

```
    for(i=0;i<n;i++)array[i]->next=NULL;
    for(i=0;i<n;i++)
        temp=T[i]%num;
        head=array[temp];哈希映射
        if(head==NULL){
            p=create();p->elem=T[i];p->count=1;p->next=NULL;
            array[temp]->next=p;
        }
       else{
            p=findinlink(head,T[i]);在制定链表中查询元素 T[i];
            if(p==NULL){
                p=create();p->elem=T[i];p->count=1;p->next=NULL;
                addtolink(head,p);
             }
             else{
                    p->count++;找到该元素
                    if(p->count>max)max=p->count;
             }
        }
}
```

19、 设计一个时间复杂性不超过 $O(n^2)$的算法，找出 n 个数组成的序列中最长的单调递增序列。

```
  typedef struct len
{
    int start;
    int end;
    int lon;
}Len;

void longest(int a[])
```

```
{
    Len first,second;
    int i,j;
    first.lon=0;
    i=0;
    while(1)
    {
        second.start=i;
       second.lon=1;
       while(a[i]<a[i+1])
       {
            second.lon++;
          if(i+1==length-1)
          {
             second.end=i+1;
             goto out;
          }
          i++;
       }
       second.end=i;
       if(second.lon>first.lon)
       {
            first.start=second.start;
          first.end=second.end;
          first.lon=second.lon;
       }
       i++;
  }
   out:
   if(first.lon<second.lon)
```

```
    {
        first.start=second.start;
       first.end=second.end;
        first.lon=second.lon;
    }

    /*输出 first 的结果*/
}
```

20、 从 1，2，…，n 中找出所有质数，设计一个比较优的算法。

```
int B[n];
void fun(){
   int i,j,k,s0=0,s1=0;
   B[0]=2;
   B[1]=3;
   k=1;
   for(i=6;i<=n;i+=6){
       for(j=1;j<=k;j++){
           if(s0==0&&(i-1)%B[j]==0)s0=1;
           if(s1==0&&(i+1)%B[j]==0)s1=1;
           if(s1==1&&s0==1)break;
       }
       if(s0==0){k++;B[k]=i-1;}
       if(s1==0){k++;B[k]=i+1;}
   }
   for(i=0;i<=k;i++){
        printf("%d ",B[i]);
        if((i+1)%10==0)printf("\n");
   }
   printf("\n");
}
```

21、 测试一个图是否连通图，可以用深度优先搜索算法或宽度优先搜索算法，在什么情况下，用哪种算法更好一些？

22、 试求图中所有点对之间的所有路径，并找出两点之间最短的路径。

23、 二叉检索树的主要问题是树高，编写一个控制检索树树高的算法。

24、 试写一个在 2—3 树上删除一个节点的算法。

25、 采用深度优先算法找出一个无向图的所有双向连通分支。

26、 编写一个算法，找出一个有向图的所有强连同分支。

```
int N=4;
int sign=0;
int visited[100]={0};
int f[100];
int cout=0;
void DFS(int v,int s){
    int j;
    visited[v]=1;sign++;
    f[cout]=v;cout++;                          //访问节点加入 f[n]
    if(s==1)printf("%d ",v);
    for(j=0;j<N;j++){
        if(G[v-1][j]==1){                      //如果 G[v-1][j]有通路且未被访问者访问之
            if(!visited[j+1]) {
                if(s==0)DFS(j+1,0);
                else DFS(j+1,1);
            }
        }
    }
}
```

```
void fun(){
    int v,i,j,t;
    for(v=0;v<N;v++)if(!visited[v+1])DFS(v+1,0); //第一次深度优先遍历
    for(i=0;i<N;i++){                                    //矩阵转置
        for(j=i+1;j<N;j++){                              //N 为图中节点数
            t=G[i][j];
            G[i][j]=G[j][i];
            G[j][i]=t;
        }
    }
    for(i=0;i<N;i++)visited[i+1]=0;                      //清空标志
    for(i=cout-1;i>=0;i--)if(!visited[f[i]]){
        DFS(f[i],1);
        if(i!=cout-1)printf("\n");
    }
}
```

27、 为什么查找连通分支的算法总是采用深度优先算法，采用宽度优先算法可以吗？为什么？

28、 编写找出一个网络的最大流的算法。

```
#include <iostream>
#include <queue>

using namespace std;

const int maxN=201;

static int edge[maxN][maxN];
bool visited[maxN];
int father[maxN];
int N, M; //边数,顶点数
int ans; //结果
void Ford_Fulkerson( )
{
```

```
while(1)
{
  //一次大循环,找到一条可能的增广路径
  queue <int> q;
  memset(visited, 0, sizeof(visited));
  memset(father, -1, sizeof(father));
  int now;
  visited[0] = true;
  q.push(0);
  while(!q.empty())//广度优先
  {
    now = q.front();
    q.pop();
    if(now == M-1) break;
    for(int i = 0; i < M; i++)
    {
      //每次父亲节点都要更新,权值减为 0 的边就不算了.
      if(edge[now][i] && !visited[i])
      {
        father[i] = now;
        visited[i] = true;
        q.push(i);
      }
    }
  }
  //可能的增广路不存在了
  if(!visited[M-1]) break;
  int u, min = 0xFFFF;
  for(u = M-1; u; u = father[u])//找出权值最小的边
  {
    if(edge[father[u]][u] < min)
      min = edge[father[u]][u];
  }
  //减去最小权值
  for(u = M-1; u; u = father[u])
  {
    //前向弧减去
    edge[father[u]][u] -= min;
    //后向弧加上
    //存在圆环,这句话关键
    edge[u][father[u]] += min;
  }
  //当前增广路径增加的流
```

```
            ans += min;
        }
    }

    int main()
    {
        int s, e, w;
        while(cin >> N >> M)
        {
            ans = 0;
            memset(edge, 0, sizeof(edge));
            for(int i = 0; i<N; i++)
            {
                cin >> s >> e >> w;
                edge[s-1][e-1] += w;
            }
            Ford_Fulkerson();
            cout << ans << endl;
        }
        return 0;

    }
```

29、 采用最大流算法编写一个二分图的最大匹配算法。

```
#include "stdafx.h"
#include <iostream>
#define N 6
#define L 3
#define R 3
using namespace std;
bool RToL(int);
int map[N][N] = {{0,0,0,1,1,1},
                 {0,0,0,0,0,1},
                 {0,0,0,0,0,1},
                 {0,0,0,0,0,0},
                 {0,0,0,0,0,0},
                 {0,0,0,0,0,0}};

int lien[N] = {-1};
int matched[N] = {0};

bool LToR(int curPoint)//左边的点找右边可以连接的点
{
    for(int i=L; i<N; i++)//对每个右边的点
    {
```

```
            if(map[curPoint][i] && lien[i] == -1)//如果未进入链里
            {
                lien[curPoint] = i;//这个点连接到右边的点 i 号
                if(RToL(i))//如果找到了增广链
                    return true;
                else//如果没找到，那就试下下个链路
                {
                    lien[curPoint] = -1;
                    continue;
                }
            }
        }
        return false;
}
bool RToL(int curPoint)
{
        for(int i=0; i<L; i++)//对每个左边的点
        {
            if(map[i][curPoint] && lien[i] == -1)
            {
                lien[curPoint] = i;
                if(LToR(i))
                    return true;
                else
                {
                    lien[curPoint] = -1;
                    continue;
                }
            }
        }
        //右边寻找的话是有特殊情况的，那就是如果没找到接下去的点的
        //而右边的点本身没匹配过那么就说明这条增广链到次为止
        for(int j=0; j<L; j++)
            if(matched[j] == curPoint+1)
                return false;
        return true;
}
void match()
{
        for(int i=0; i<L; i++)
        {
            memset(lien, -1, sizeof(lien));
            if(matched[i]==0)//左边的开始节点必须为未配对的
            if(LToR(i))//寻找
```

```
            {
                for(int g=0; g<L; g++)//只对每个左边的点查看
                {
                    if(lien[g]!=-1 && matched[lien[g]] == 0)//如果左边的点有连接
                    {
                        matched[matched[g]-1] = 0;//把原本右边可能已经配对的
去掉配对
                        matched[g] = lien[g]+1;//然后从新连接左边到右边
                        matched[lien[g]] = g+1;//然后把右边的连接到左边
                    }
                }
            }
        }
        for(int j=0; j<N; j++)
            cout<<"第"<<j+1<<"号顶点配:"<<matched[j]<<"号顶点 "<<endl;
}
int main()
{
    match();
    return 0;
}
```

30、 什么是“可证难解性”问题，试举出两个例子。

“可证难解性”问题是不可判定的难解问题和可判定的难解问题的总称，比如货郎担问题，子图同构问题。

31、 P 类、NP 类、NPC。

P 类可以找到一个能在多项式的时间里解决它的算法

NP 问题是指可以在多项式的时间里验证一个解的问题

NPC 如果任何 NP 问题都能通过一个多项式算法转换为某个 NP 问题，那么这个 NP 问题就称为 NPC 问题。

32、 一个问题的“难度”是该问题固有的，试举例说明。

一个问题难解性是问题固有的性质，多项式算法是划分问题的类的标准。如果一个问题是不可用的多项式算法求解的，则该问题是难解的。

如停机问题、平铺问题。

33、 简述 DTM、NDTM 的工作过程。

DTM 的执行过程如下:

1.开始时，将 *中的字符串 ω 放在从 1 到 ω 的方格中，其它地方放空格符，控制器处在 q0 状态。

2.读写头扫描第一个方格中的符号 a0，转移函数 σ(q0,a0)= (q1,a1,△)根据事先

设计好的 σ 函数表

将当前状态改变为 q1

在当前格中将 a0 改写为 a1

根据△=R,S, L 确定读写头向右(R),向左(L) ，或不动(S)

(一般为 σ(qi,ai) = (qi+1,ai+1,△))

3.直至当前状态 q 为 qy 或 qn,计算停止，

若 q= qy,答案是“肯定”,DTM 接受 ω；

若 q= qn,答案是“否定”,DTM 拒绝 ω。

4.每个步骤即为一步计算。

非确定型单带图另机（NDTM），其结构与 DTM 基本相同，只是多了一个猜想模块和一个只写头。 NDTM 形式地定义为（Q， ，Γ，σ， ，q0 ，Qf）

34、 为什么说，只要两个不同的编码系统是多项式相关的，就不影响算法复杂性的多项式性。

某问题在 e1 编码下的算法时间复杂性是多项式 T(n)。在 e2 编码下，算法不变，算法时间复杂性 T(P(n))，也是一个多项式。

35、 简述 SAT 问题和 COOK 定理。

SAT 问题：给定一组布尔变量 V 和一组由 V 组成的字句集合 C，判定是否存在一组满足 C 中所有子句的真值赋值。（布尔表达式可满足性问题）

布尔表达式的可满足性问题 SAT 是 NP 完全的。

36、 如何证明一个问题是 NPC 的。已知 TSP 是 NPC 的，试证明 Hamilton 问题也是 NPC 的。

先证明它至少是一个 NP 问题，再证明其中一个已知的 NPC 问题能约化到它（由约化的传递性，则 NPC 问题定义的第二条也得以满足；至于第一个 NPC 问题是怎么来的，下文将介绍），这样就可以说它是 NPC 问题了

同样地，我们可以说，Hamilton 回路可以约化为 TSP 问题(Travelling Salesman Problem，旅行商问题)：在 Hamilton 回路问题中，两点相连即这两点距离为 0，两点不直接相连则令其距离为 1，于是问题转化为在 TSP 问题中，是否存在一条长为 0 的路径。Hamilton 回路存在当且仅当 TSP 问题中存在长为 0 的回路。

37、 当前计算机的速度越来越高，为什么还要研究时间复杂性更低的算法？

问题的复杂过程和规模的线性增长导致时耗的增长和空间需求的增长对低效的算法来说是超线性的，绝非计算机的速度和容量的线性增长得来的时耗减少和

存储空间的扩大所能抵消的。