第十一章

计算机不是万能的

我们的问题分类法

目前计算机已用来求解各类问题，但是其能力不是没有极限的，因此计算机并不能用来求解所有的问题，“什么能计算”是计算机科学的根本问题。

首先问题可以分为计算机能够求解和不能求解，这一划分基于问题是否能够在有限时间内解决，与计算时间长短无关。

在计算机能够求解的问题中，以是否能够设计出多项式时间算法为分界线将其分为易解的和难解的两类，对于难解的问题，虽然可以求解，可以为其设计算法，但由于算法的时间复杂性为指数型，不具有实用价值，并且这些问题目前还设计不出多项式时间的算法，将来也不大可能发现多项式时间的算法。

不可计算的问题

场景：计算机正在执行一个程序，我们坐在边上等待程序结束。

问题：当过了很久程序还没结束时，我们该怎么办呢?

1）推测程序中可能出现了无穷循环，永远不会结束，这样我们就必须强行中断程序运行甚至重启计算机。

2）也许是因为计算太复杂导致时间过长呢？这样的话，我们就该继续等待。

如何选择?

设想：要是有这么一个程序P，其功能是以另一个程序Q的代码作为输入，并分析Q中是否包含无穷循环。

很遗憾，这样的程序P是不存在的!

这就是所谓停机问题（Halting problem）。

停机问题

从算法设计角度看，停机问题就是要设计算法halt，它的输入为另一个程序prog的源代码，输出为prog是否无穷循环。由于prog的行为不仅依赖于它的源代码，还依赖于它的输入数据，因此为了分析prog的终止性，还要将prog的输入数据data交给halt。由此可得halt的定义说明：

算法：停机判别算法halt

输入：程序prog的源代码，以及prog的输入数据data，都以字符串形式表示

输出：如果prog在data上的执行能终止，则输出True，否则输出False

在停机问题中，正常情况下是想运行输入为data的程序prog，即prog(data)，但又不知道这个执行过程能不能终止，于是希望将prog的代码和data交给停机分析算法halt，由halt来判断prog(data)的终止性。

停机问题

假如已经设计出了停机分析算法halt，虽然不知道halt的实现过程，但基于halt算法设计如下算法strange

```
算法 strange
输入：字符串p
    1. result = halt(p,p)
    2. if result == True    {halt算法判定p(p)终止}
    3.    while True
    4.        pass
    5.    end while
    6. else             {halt算法判定p(p)不终止}
    7.    return
    8. end if
```

那么运行strange(strange)的结果是什么呢?

停机问题

strange首先调用halt(p,p)，这里的关键是传递给halt的两个输入都是p，亦即要分析程序p以它自己为输入数据时——即p(p)——运行是否终止。strange根据halt(p,p)的分析结果来决定自己接下去怎么做：如果结果为True，即p(p)能终止，则strange进入一个无穷循环；如果结果为False，即p(p)不终止，则strange就结束。

strange程序看上去有点费解，但只要halt存在，strange在编程方面显然没有任何问题。

将strange自身的源代码输入给strange时会发生什么？更确切地，strange(strange)能否终止?

停机问题

参照上面的strange代码来分析。假如调用strange(strange)不终止，那必然是因为执行到了代码中条件语句的if result == True部分，即halt(strange,strange)返回了True，这又意味着strange以strange为输入时运行能终止！另一方面，假如调用strange(strange)能终止，那必然是因为执行到了条件语句的else部分，即halt(strange,strange)返回了False，这又意味着strange以strange为输入时运行不能终止！总之，我们得到了如下结论：

若strange(strange)不终止，则strange(strange)终止；

若strange(strange)终止，则strange(strange)不终止。

这样的结论在逻辑上显然是荒谬的。导致这个矛盾的原因在于假设了halt的存在，并利用了halt的功能

停机问题是一个不可解问题，即没有一个计算机程序能够确定另外一个计算机程序是否会对一个特定的输入停机。

对于可以求解的问题，
又会有什么情况出现呢?

“I can’t find an efficient algorithm, I guess I’m just too dumb.”

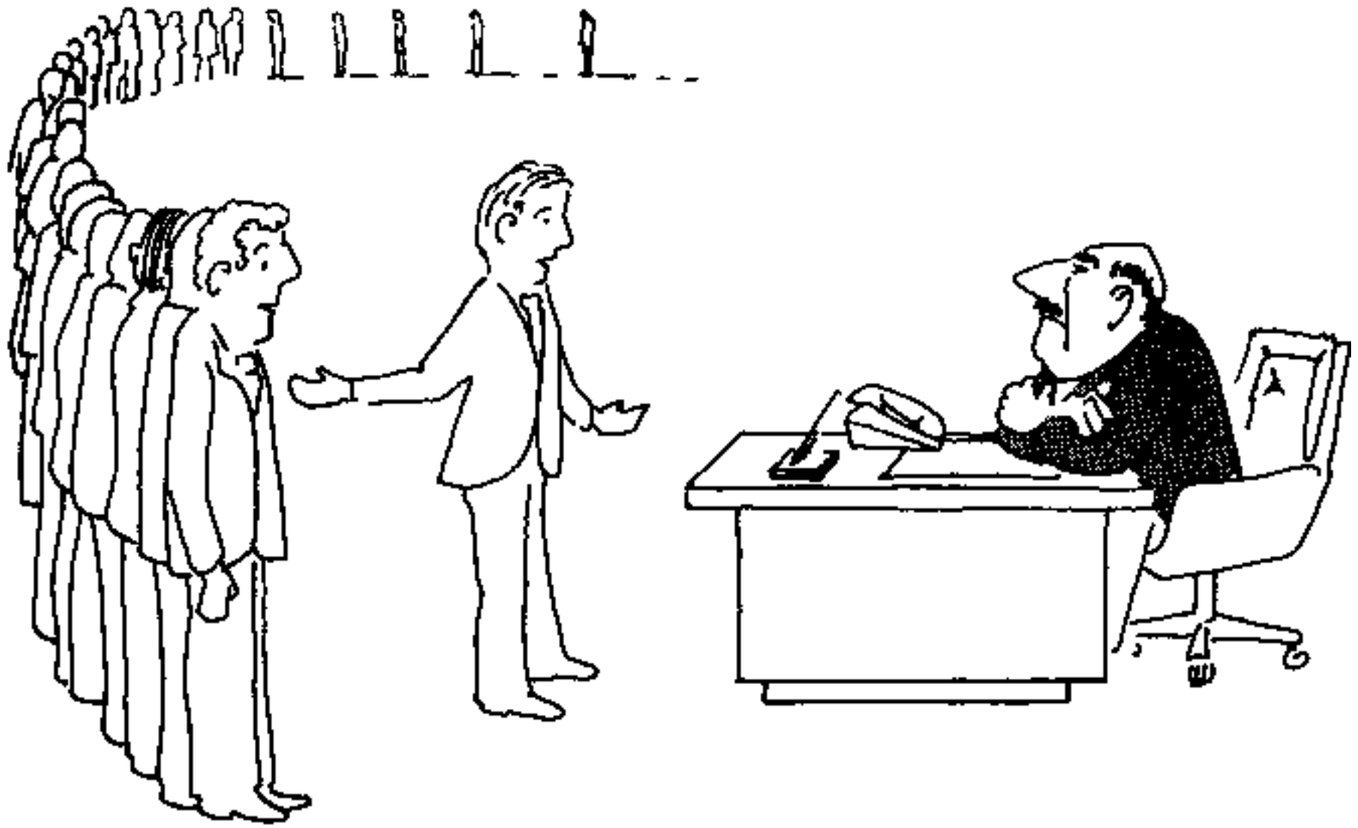
Serious damage to your
position within the company !!!

“I can’t find an efficient algorithm, because no such algorithm is possible!”

Unfortunately, proving intractability can be just
as hard as finding efficient algorithms !!!

∴ No hope !!!

P ≠ NP



“I can’t find an efficient algorithm, but neither can all these famous people”

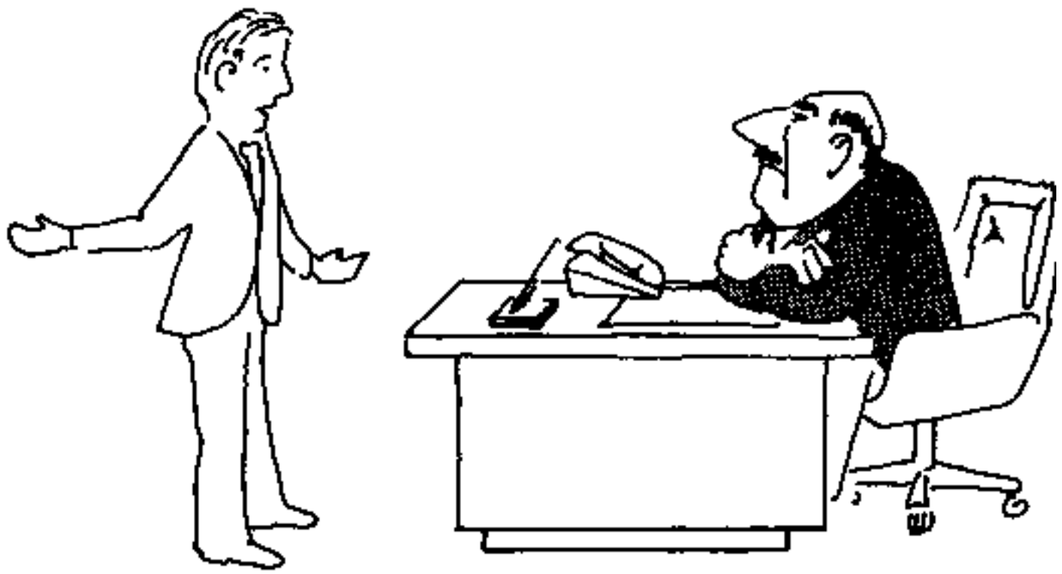
Comparison of several polynomial and exponential time complexity functions

Time complexity function	Size n					
	10	20	30	40	50	60
n	.00001 second	.00002 second	.00003 second	.00004 second	.00005 second	.00006 second
n^2	.0001 second	.0004 second	.0009 second	.0016 second	.0025 second	.0036 second
n^3	.001 second	.008 second	.027 second	.064 second	.125 second	.216 second
n^5	.1 second	3.2 second	24.3 second	1.7 minutes	5.2 minutes	13.0 minutes
2^n	.001 second	1.0 second	17.9 minutes	12.7 days	35.7 years	366 centuries
3^n	.059 second	58 minutes	6.5 years	3855 centuries	$2*10^8$ centuries	$1.3*10^{13}$ centuries

能解.但时间为无穷.

Size of Largest Problem Instance Solvable in 1 Hour

Time complexity function	With present computer	With computer 100 times faster	With computer 1000 times faster
n	N_1	$100N_1$	$1000N_1$
n^2	N_2	$10N_2$	$31.6N_2$
n^3	N_3	$4.64N_3$	$10N_3$
n^5	N_4	$2.5N_4$	$3.98N_4$
2^n	N_5	$N_5+6.64$	$N_5+9.97$
3^n	N_6	$N_6+4.19$	$N_6+6.29$



“Anyway, we have to solve this problem.
Can we satisfy with a good solution ? “

一般的观点

运行时间与输入规模的某个多项式函数相关，一般称这样的算法是高效的(efficient)，相应的问题是易处理的(tractable)。换句话说，如果算法的运行时间是O(P(n))，这里P(n)是输入规模n的多项式函数，那么可称该算法是高效的。用P（指多项式时间）表示所有可以被高效算法所解决的问题类。

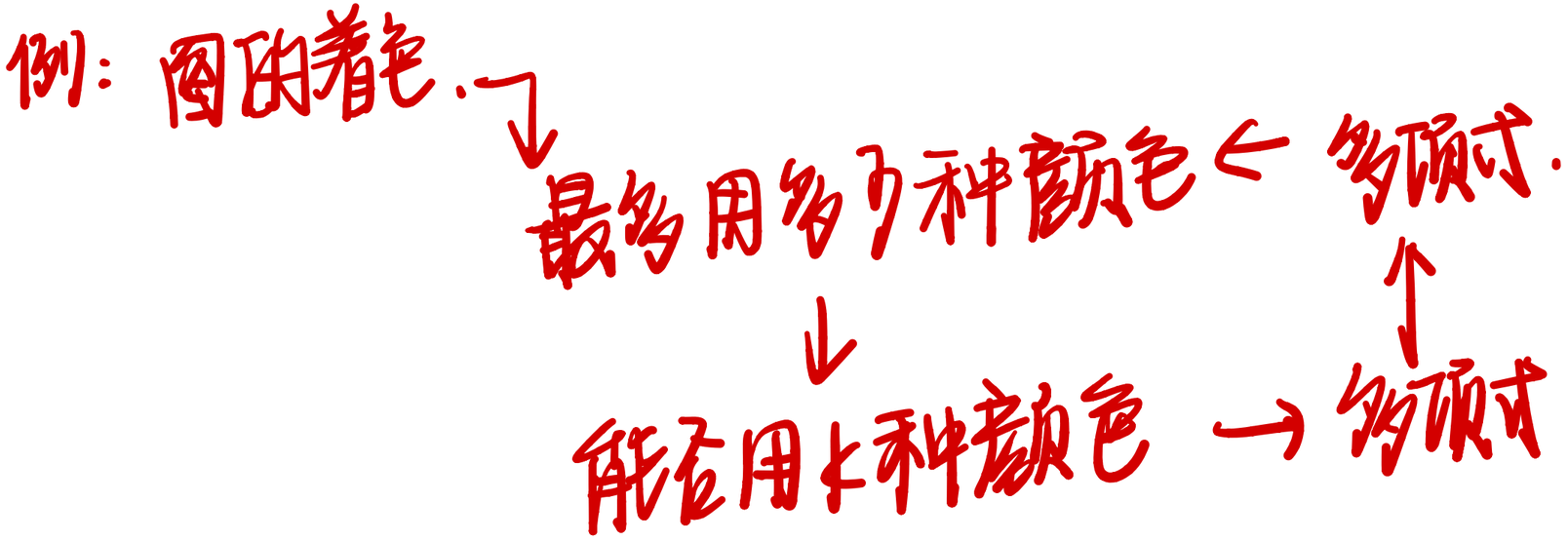
NP完全问题不存在高效算法

判定问题

判定问题（decision problem），即只考虑那些答案为肯定或者否定的问题。

大多数问题都可以转化成判定问题。

判定问题可以视为语言识别问题，假设U是由该判定问题的各种可能的输入所组成的集合，而$L \subseteq U$是回答为肯定的全部输入所组成的集合，一般称L为与该问题相关的语言。



多项式归约

设L_1和L_2分别是输入空间U_1和U_2的两个语言，如果存在多项式时间算法可以将每个输入$u_1 \in U_1$转化成另一个输入$u_2 \in U_2$，使得$u_1 \in L_1$当且仅当$u_2 \in L_2$，则称L_1可以多项式归约到L_2。该算法是输入u_1的大小的多项式函数，同时u_2的大小是u_1的大小的多项式函数。

如果有L_2的算法，就可将这两个算法组合起来生成L_1的算法。

定理11.1：如果L_1可以多项式归约到L_2并且L_2有一个多项式时间的算法，那么L_1也存在一个多项式时间的算法。

归约的概念并不是对称的

如果两个语言L_1和L_2可以相互多项式归约到对方，则称它们是多项式等价的，或等价的。

定理11.2：如果L_1可以多项式归约到L_2并且L_2可以多项式归约到L_3，那么L_1可以多项式归约到L_3。

非确定算法

除具备确定性算法的所有常规操作外，非确定算法有一个非常强的基本操作，称为nd-选择。该基本操作与固定数目的一些可选项结合在一起使用，每次进行选择，从而算法可沿着不同的路径进行。给定输入x，非确定算法交替执行常规的确定性步骤和nd-choice操作，最终决定是否接受x。

确定性和非确定性算法最突出的差异：识别语言的方式。

一个非确定性算法能识别语言L是指给定输入x，$x \in L$当且仅当可以将算法执行过程中的每一次nd-选择转变成确定的选择，使得算法能接受x。

对于输入$x \in L$，算法的运行时间是指到达接受状态的最短执行序列长度，非确定算法的运行时间是指对于所有$x \in L$在最坏情况下的运行时间

图的完美匹配.

for (all edges)
随机决定e是否加入匹配M
检查M是否完美

NP

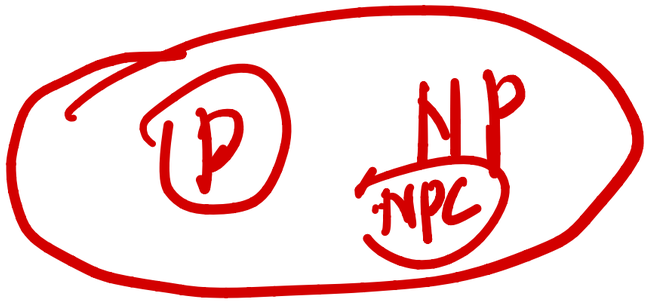
存在运行时间关于输入是多项式函数的非确定算法的问题所组成的类称为NP类

决定P和NP之间关系的问题称为P=NP问题。

定义：如果所有NP中的问题都可以多项式归约到问题X，则称X为NP难（NP－hard）问题 。

定义：如果问题X满足（1）X在NP中，（2）X是NP难的，那么称X为NP完全问题。

引理11.3：如果问题X满足（1）X属于NP，（2）对于某个NP完全问题Y，Y可以多项式归约到X，那么X是NP完全问题。



$P \subseteq NP$

NPC → P=NP

可满足性问题

设S是一个用合取范式（CNF，Conjunctive Normal Form） 表示的布尔表达式，也就是说S是一些和（或）的积（与），如果存在使整个表达式取值为1的布尔变量赋值， 则称该布尔表达式是可满足的。

SAT问题：判定所给定的表达式是否可满足（不必找到使之满足的赋值）。

可以猜测一个真值指派并且在多项式时间内验证它是否满足表达式，故SAT问题在NP中。

Cook定理：SAT问题是NP完全的。

NP完全性的证明

要证明一个新问题是NP完全的，首先要证明其属于NP，这通常是（但并非总是！）容易的，接着要将一个已知的NP完全问题在多项式时间内归约到这个新问题。

(最小)顶点覆盖问题(VC)

设G=(V,E)是一个无向图，G的顶点覆盖是一个顶点集合，满足G中所有的边都至少和该集合中的一个顶点相关联。

问题：给定无向图G=(V,E)和一个整数k，判定G是否有包含≤k个顶点的顶点覆盖。

定理11.4：顶点覆盖问题是NP完全的。

将团问题归约到顶点覆盖问题

① 为NP的（随机取顶点）

② ∀团集问题的实例：$G=(V,E)$，$k\geq 0$.

VC的实例 → $\bar{G}=(V,\bar{E})$是G的补图

n-k

① 团集问题有解 $C=(U,F)$

$|U|\geq k$ $V-U$是$\bar{G}$的VC

$|V-U|\leq n-k$

$\forall (u,v)\in\bar{E}$. $(u,v)\notin E$

即u或v至少一个属于V-U

∴ (u,v)被u-v覆盖

C有解 → VC有解

②若D是$\bar{G}$的VC，$|D| \leq n-k$

D与$\bar{G}$中所有的边关联

$(u,v) \in \bar{E}$

VC有解→C有解.

支配集问题

设G=(V,E)是一个无向图，如果顶点集合D满足，G中的所有顶点要么在D中要么与D中至少一个顶点相邻，则称D为支配集。

问题：给定无向图G=(V,E)和整数k，判定G中是否有一个包含≤k个顶点的支配集。

定理11.5：支配集问题是NP完全的。

将顶点覆盖问题归约到支配集问题

$\forall G=(V,E)$ $\forall (v,w)\in E$

增加顶点 u. 边 (v,u), (u,w) $\rightarrow$ G', k

有向图

3SAT问题

3SAT问题是一般SAT问题的简化，3SAT的实例是指每个子句中恰好含有3个变量的布尔表达式。

问题：给定以CNF形式出现并且每个子句恰好含有3个变量的布尔表达式，判定其是否可满足。

定理11.6：3SAT问题是NP完全的。

一般SAT问题归约到3SAT问题

团问题

给定无向图G=(V,E)，G中的一个团C是G的一个子图，满足C中的任何两个顶点均相邻，
　　换句话说，团即完全子图。

问题：给定无向图G=(V,E)和整数k，判定G是否包含一个大小≥k的团。

定理11.7：团问题是NP完全的。

将SAT问题归约到团问题

3着色问题

设G=(V,E)是无向图，G的有效着色是指对所有顶点的颜色指派，使得每个顶点被指派一种颜色并且相邻顶点不被指派成相同颜色。

问题：给定无向图G=(V,E)，判定G是否可以被3种颜色着色。

定理11.8：3着色问题是NP完全的。

将3SAT问题归约到3着色问题

一般的经验

首先，证明Q属于NP，这通常是（但非总是）简单的。接着要选择一些看上去与Q相关或者类似的已知的NP完全问题，但有时问题看上去差异很大，因此很难选择“类似”的目标，这只能通过经验来把握，一般的方法是尝试对几个问题归约，直到发现一个成功的例子为止。

归约是从一个已知NP完全问题的任意实例到问题Q，通常最容易犯的错误是反方向进行归约

在归约过程中可以有一定的自由度：Q包含的参数可以将它设置成任何固定的值；只要归约可以在多项式时间完成，其效率并不重要。不仅可以忽略常数因子（例如，使问题规模增加一倍），还可以使问题规模扩大平方倍！

通用的技术：证明一个NP完全问题是Q的特例是最简单的；局部归约

其它常见NP完全问题

哈密尔顿问题：给出无向图G=(V,E)，是否存在一条遍历每个顶点一次且仅一次的路径?

旅行商问题：给出加权无向图G=(V,E)，求出遍历每个顶点一次且仅一次的最短路径

0-1背包问题：$U=\{u_1, u_2,..., u_n\}$是一个准备放入容量为C的背包中的n个物品的集合，第i个物品u_i具有体积s_i和价值v_i，要求从这n个物品中挑选出一部分装入背包，在不超过背包容量的前提下使背包中物品的价值最大。

装箱问题：给出大小为$s_1, s_2, \ldots, s_n$的n个物品，能否最多用k个容量为C的箱子将这些物品装入?

集合覆盖问题：给定集合X以及X的子集族F，是否存在F中的k个子集，它们的并集是X?

作业调度问题：有n个作业$J_1, J_2, \ldots, J_n$，每项作业J_i的需要的机器运行时间为t_i，那么能否在时间T内利用m台机器完成所有n项作业?

处理NP完全问题的技术

NP完全问题理论能识别那些不太可能存在多项式时间算法的问题，但如何解决?

识别出其中能够多项式时间求解的特例，如2SAT问题就属于P类问题；

回溯法与分支限界法：这两种算法在求解问题时，虽然并不是多项式时间算法，但在解决其中不少实例时仍然会取得较快的速度；

确保一定性能的多项式时间算法：虽然其解与最优解之间存在一定差异，但却能够在多项式时间内完成，这类算法称为近似算法

随机算法：虽然不能够保证得到正确的解，但能够在多项式时间内完成。

回溯法

回溯法是一种有组织的穷举搜索方法，但经常能够在不试探所有可能性的情况下得到解，
适用问题的特性是虽然可能有很多潜在的解，但真正需要考虑的并不多。

3着色问题

潜在的解的数目为3^n，即对n个顶点进行3着色的各种可能。

除非图中没有边，有效解的数目将远小于3^n，这是因为边对着色加了限制。为了探求所有顶点的着色方案，可首先对其中一个顶点任意着色，并且在满足由边所带来的约束条件（即相邻顶点必须着色不同）的同时，继续给其他顶点着色。当对一个顶点进行着色时，可尝试各种与先前顶点的着色不冲突的方案。该过程可以通过树遍历算法来完成

```
Algorithm 3-coloring(G,var U)
Input: G=(V,E) (an undirected graph), and U (a set of vertices that have already been colored together
     with their colors)      {U is initially empty}
Output: An assignment of one of three colors to each vertex of G
begin
   if U=V then print “coloring is completed”; halt
   else
      pick a vertex v not in U;
      for C:=1 to 3 do
         if no neighbor of v is colored with color C then
            add v to U with color C;
            3-coloring(G,U)
end
```

分枝限界法

分枝限界法是回溯法求解涉及寻找某个目标函数最小（或最大）值问题的一种变形方法

0/1背包问题

假设有4个物品，其重量分别为(4, 7, 5, 3)，价值分别为(40, 42, 25, 12)，背包容量 W=10。首先，将给定物品按单位重量价值从大到小排序，结果如下：

物品	重量(w)	价值(v)	价值/重量(v/w)
1	4	40	10
2	7	42	6
3	5	25	5
4	3	12	4

0/1背包问题

求得近似解为(1, 0, 0, 0)，获得的价值为40，这可以作为0/1背包问题的下界。

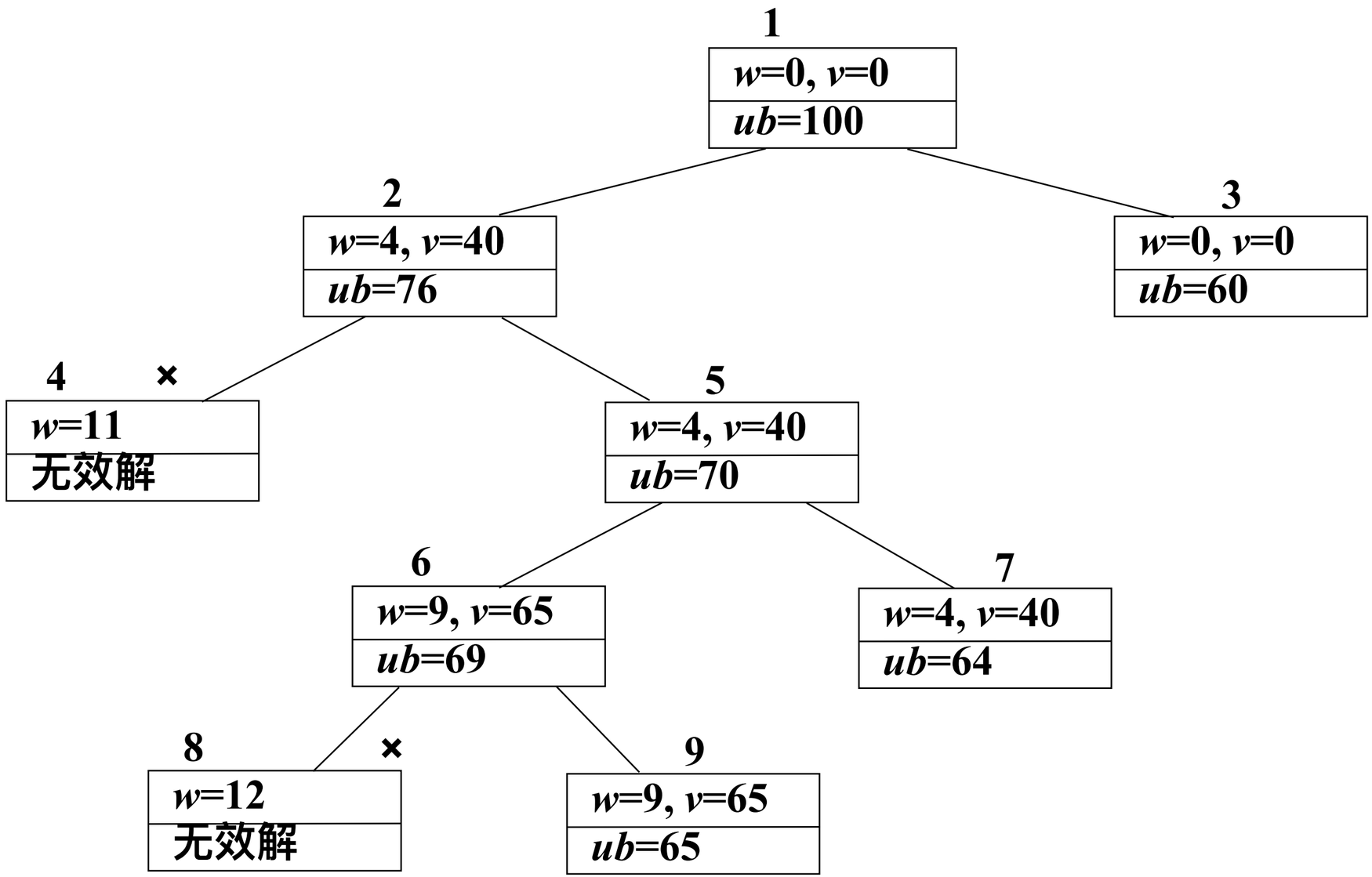
考虑最好情况，背包中装入的全部是第1个物品且可以将背包装满，则可以得到一个非常简单的上界的计算方法：$ub=W\times(v_1/w_1)=10\times10=100$。

目标函数的界：[40, 100]。

限界函数为：

$$ub = v + (W - w) \times (v_{i+1}/w_{i+1})$$

分支限界法求解0/1背包问题



搜索过程

(1) 在根结点1，没有将任何物品装入背包，因此，背包的重量和获得的价值均为0，根据限界函数计算结点1的目标函数值为10×10=100；

(2) 在结点2，将物品1装入背包，因此，背包的重量为4，获得的价值为40，目标函数值为40 + (10-4)×6=76，将结点2加入待处理结点表PT中；在结点3，没有将物品1装入背包，因此，背包的重量和获得的价值仍为0，目标函数值为10×6＝60，将结点3加入表PT中；

(3) 在表PT中选取目标函数值取得极大的结点2优先进行搜索；

搜索过程

(4) 在结点4，将物品2装入背包，因此，背包的重量为11，不满足约束条件，将结点4丢弃；在结点5，没有将物品2装入背包，因此，背包的重量和获得的价值与结点2相同，目标函数值为40 + (10-4)×5=70，将结点5加入表PT中；

(5) 在表PT中选取目标函数值取得极大的结点5优先进行搜索；

(6) 在结点6，将物品3装入背包，因此，背包的重量为9，获得的价值为65，目标函数值为65 + (10-9)×4=69，将结点6加入表PT中；在结点7，没有将物品3装入背包，因此，背包的重量和获得的价值与结点5相同，目标函数值为40 + (10-4)×4＝64，将结点6加入表PT中；

搜索过程

(7) 在表PT中选取目标函数值取得极大的结点6优先进行搜索；

(8) 在结点8，将物品4装入背包，因此，背包的重量为12，不满足约束条件，将结点8丢弃；在结点9，没有将物品4装入背包，因此，背包的重量和获得的价值与结点6相同，目标函数值为65；

(9) 由于结点9是叶子结点，同时结点9的目标函数值是表PT中的极大值，所以，结点9对应的解即是问题的最优解，搜索结束。

确保性能的近似算法

近似算法的解和最优解的差距并不大

计算时间为多项式时间

顶点覆盖问题

近似算法：假设G=(V,E)是一个图，M是G的一个极大匹配，由于M是一个匹配，其所包含的边没有共同顶点，又因为M是极大的，所有其他边至少和M中的一条边有相同顶点。

定理11.9：与极大匹配M中的边相关联的所有顶点构成一个顶点覆盖，并且其大小不超过最小顶点覆盖大小的2倍。

寻找极大匹配的方法是简单收集边直到所有边都被覆盖到。但寻找最小极大匹配（即有最少边个数的极大匹配）问题也是NP完全的

一维箱柜包装问题

箱柜包装问题是指将不同大小物体打包到规定大小的箱柜中，并且箱柜使用的数目要尽可能少。

问题：设$x_1,x_2,...,x_n$是介于0和1之间的实数，将这些数划分成尽可能少的子集使得每个子集中的数总和不超过1。

First Fit算法：将x_1放入第一个箱柜，接着对于每个i，将x_i放入能容纳它的第一个箱柜，或者当使用过的箱柜都不能容纳它时则使用一个新的箱柜。

定理11.10： First Fit算法至多需要2OPT个箱柜(常数2还可以减为1.7)。

Decreasing First Fit算法：先逆序排序，然后再使用First Fit。

定理11.11： Decreasing First Fit算法至多需要11/9OPT+4个箱柜。

欧几里德旅行商问题

问题：$C_1,C_2,...,C_n$ 是平面中的点集合，其对应于n个城市的位置；试找到一条长度最短的哈密尔顿回路

首先计算最小代价生成树（这里代价=距离），树的代价不会超过最佳TSP长度。

考察由深度优先搜索遍历树（从任意顶点出发）所形成的环路，每条边恰好被遍历了两次，所以这个环路的代价是这棵最小生成树代价的两倍，因而也不会超过最短TSP行程的两倍。可以通过选取直接路线而非回溯的方法将这个环路转变成一个TSP路径，即直接到第一个新顶点而不是回溯。由于采用欧几里德距离，这样形成的TSP路径长度不会超过最短TSP路径长度的两倍。

复杂度：运行时间主要由最小代价生成树算法的运行时间所决定的，对于欧几里德图为O(nlogn)

改进

关键：高效地将树转变成欧拉图

欧拉图中所有节点的度均为偶数，考虑树中的所有度为奇数的节点，它们的个数一定是偶数（否则所有节点度数的总和将会是奇数，由于节点度数总和恰好是边数目的2倍，因而这是不可能的）。

对这棵树添加足够多的边，使得所有节点的度均为偶数，这样就得到了一个欧拉图。由于TSP路径将由这个欧拉环路（会走一些捷径）构成，因此要将额外添加的边的长度最小化

改进

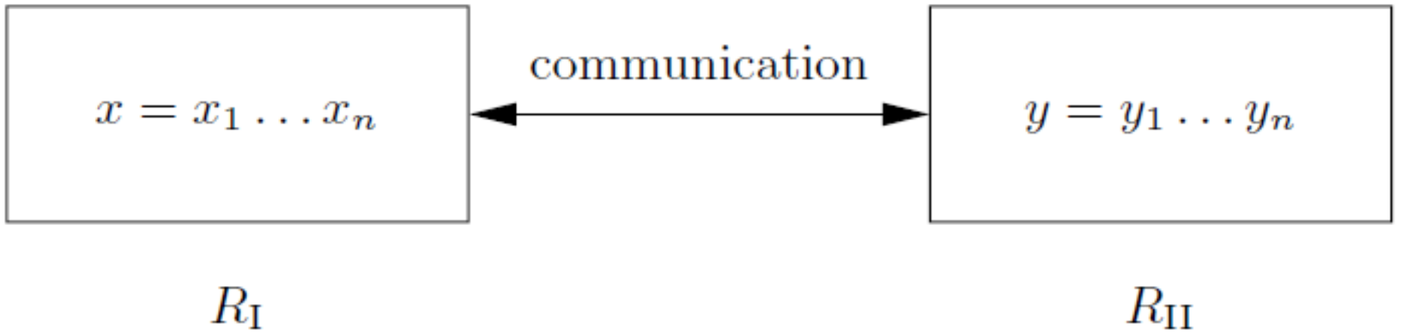
问题：给定平面中的一棵树，希望添加一些边使之成为一个欧拉图，并且要使添加边的总长最小。

对于每个度为奇数的顶点至少要添加一条边，尝试就添加一条。假定有2k个度为奇数的顶点，如果添加k条边，每条边连接两个度为奇数的顶点，那么所有边的度都为偶数了。这样该问题就变成了匹配问题了，并且要找到一个最短长度匹配，使其覆盖所有度为奇数的顶点。对于一般图可以在$O(n^3)$时间内找到权值最小的完美匹配，最近提出了一个针对欧几里德距离的算法，运行时间为$O(n^{2.5}(\log n)^4)$。最终的TSP路径可以通过走捷径的方法从欧拉图获得

定理11.12：改进后算法所生成的TSP行程，其长度至多为最短TSP行程长度的1.5倍。

随机算法

考虑如下场景：



需要判断R_I 和 R_{II} 的数据是否相同
需要多少通信量?
显然任何一种确定性算法需要在R_I and R_{II}之间传输至少n比特
若$n = 10^{16}$，则要将这些数据全部正确地传送并不是一件容易的小事

算法

设$x = x_1x_2 \ldots x_n, x_i \in \{0, 1\}$，记

$$\text{Number}(x) = \sum_{i=1}^{n} 2^{n-i} \cdot x_i$$

初始状况：R_I有n位序列$x = x_1x_2 \ldots x_n$，R_{II}有n位序列$y = y_1y_2 \ldots y_n$.

1：R_I 在$[2, n^2]$ 内随机选择素数p

2：R_I计算得到整数$s = \text{Number}(x) \bmod p$，将s和p的二进制表示发送给$R_{II}$

3：收到s和p之后，R_{II} 计算$q = \text{Number}(y) \bmod p$.

若$q = s$，R_{II} 输出 “x = y”

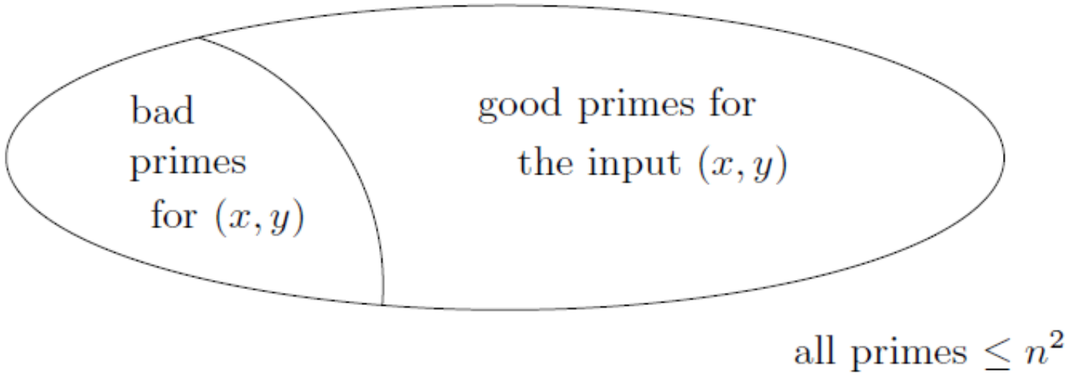
若$q \neq s$，R_{II} 输出 “$x \neq y$”

算法的性能

所需通信量仅为4logn

但这种算法可能出错

对任意的(x, y)，相应的坏素数的个数最多n-1，因此对于输入(x,y) (x≠y)，
错误率最多为(ln n^2)/n



奇妙的结论

这一错误率还可以通过重复运行降低

理论上，所有的确定性算法必然得到正确的结果，而随机算法却可能出错

但是实践中确定性算法不一定就能够得到正确结果，计算机硬件可能会出错，程序运行时间越长，硬件出错的可能性越大.

结论：一个快速的随机算法会比一个慢速的确定性算法更可靠