第五章

基于归纳的算法设计

算法设计的基本思想----从简单到复杂

将复杂的待求解问题变成简单问题有两种途径

1）对问题进行横向分解，将原问题分解为一系列简单的互不相同的子问题，即模块化设计。

2）将问题进行纵向分解，即降低问题的规模。

原理

(1) 解决问题的一个小规模事例是可能的(基础事例)

(2) 每一个问题的解答都可以由更小规模问题的解答构造出来(归纳步骤)。

关键：如何简化问题。

归纳（自递归）

$p(n) - p(n-1)$

分治

$p(n) - p(n/2)$

动态规划.

多项式求值

问题：给定一串实数a_n，a_{n-1}，…，a_1，a_0，和一个实数x，计算多项式$P_n(x)=a_nx^n+a_{n-1}x^{n-1}+\ldots+a_1x+a_0$的值。

归纳假设：已知如何在给定a_{n-1}，…，a_1，a_0和点x的情况下求解多项式(即已知如何求解$P_{n-1}(x)$)。

$P_n(x)= P_{n-1}(x)+a_nx^n$

加法：$A(n)=A(n-1)+1=n$

乘法：$M(n)=M(n-1)+n=\frac{n(n+1)}{2}$

$\sim n^2$

需要$n(n+1)/2$次乘法和n次加法运算。

观察：有许多冗余的计算，即x的幂被到处计算。

更强的归纳假设：已知如何计算多项式$P_{n-1}(x)$的值，也知道如何计算x^{n-1}。

计算x^n仅需要一次乘法，然后再用一次乘法得到a_nx^n，最后用一次加法完成计算，总共需要2n次乘法和n次加法。

利用前后两项的关系计算后项

$Q_n(x)=Q_{n+1}(x)\cdot x$　　$M(n)=M(n-1)+2=2n$

多项式求值

归纳假设(翻转了顺序的)：已知如何计算$P'_{n-1}(x)=a_nx^{n-1}+a_{n-1}x^{n-2}+\ldots+a_1$。
$P_n(x)=xP'_{n-1}(x)+a_0$。所以，从$P'_{n-1}(x)$计算$P_n(x)$仅需要一次乘法和一次加法。
该算法仅需要n次乘法和n次加法，以及一个额外的存储空间。

窍门是很少见的从左到右地考虑问题的输入，而不是直觉上的从右到左。另一个常见的可能是对比自上而下与自下而上(当包含一个树结构时)。

$P_0(x)=a_n$
$P_1(x)=a_nx+a_{n-1}$
$\vdots$
$P_k(x)=a_nx^k+a_{n-1}x^{k-1}+\cdots+a_{n-k}$

$P_{n-1}(x) \rightarrow P_n(x)$ 只有1次乘法和一个加法.

最大导出子图

令G=(V，E)是一个无向图。一个G的导出子图是一个图H=(U，F)，满足U⊆V且U中两顶点若在E中有边则该边也包含在F中。

问题：给定一个无向图G=(V，E)和一个整数k，试找到G的一个最大规模的导出子图H=(U，F)，其中H中所有顶点的度≥k，或者说明不存在这样的子图。

解决问题的一个直接方法是把度<k的顶点删除。当顶点连同它们连接的边一起被删除后，其它顶点的度也可能会减少。当一个顶点的度变成<k后它也会被删除。但是，删除的次序并不清楚。我们应该首先删除所有度<k的顶点，然后再处理度减少了的顶点呢？还是应该先删除一个度<k的顶点，然后继续处理剩下受影响的顶点？

最大导出子图

任何度<k的顶点都可以被删除。删除的次序并不重要。这种删除是必须的，而删除后剩下的图必定是最大的

设 $|V|=n$

① $n \le k$ ×

② $n=k+1$ 完全图满足

③ $n>k+1$

寻找一对一映射

问题：给定一个集合A和一个从A到自身的映射f，寻找一个元素个数最多的子集 $S \subseteq A$，S满足：

(1) f把S中的每一个元素映射到S中的另一元素(即，f把S映射到它自身)，

(2) S中没有两个元素映射到相同的元素(即，f在S上是一个一对一函数)。

寻找一对一映射

归纳假设：对于包含n-1个元素的集合，如何求解问题是已知的。

假定有一个包含n个元素的集合A，并且要寻找一个满足问题条件的子集。我们断言，任何没有被其它元素映射到的元素i，不可能属于S。否则，如果$i \in S$且S有k个元素，则这k个元素映射到至多k-1个元素上，从而这个映射不可能是一对一的。如果存在这样的一个i，则我们简单地把它从集合中删除。现在我们得到集合$A/=A-\{i\}$，其元素个数为n-1，由归纳假设，我们已知对$A/$如何求解。如果不存在这样的一个i，则映射是一对一的，即为所求，结束。

映射算法

```
Algorithm Mapping(f,n)
Input: f (an array of integers whose values are between 1 to n)
Output: S (a subset of the integers from 1 to n, such that f is one-to-one on S)
begin
   S:=A;    {A is the set of numbers from 1 to n}
   for j:=1 to n do c[j]:=0;
   for j:=1 to n do increment c[f[j]];
   for j:=1 to n do
      if c[j]=0 then put j in Queue;
   while Queue is not empty
      remove i from the top of the queue;
      S:=S-{i};
      decrement c[f[i]];
      if c[f[i]]=0 then put f[i] in Queue
end
```

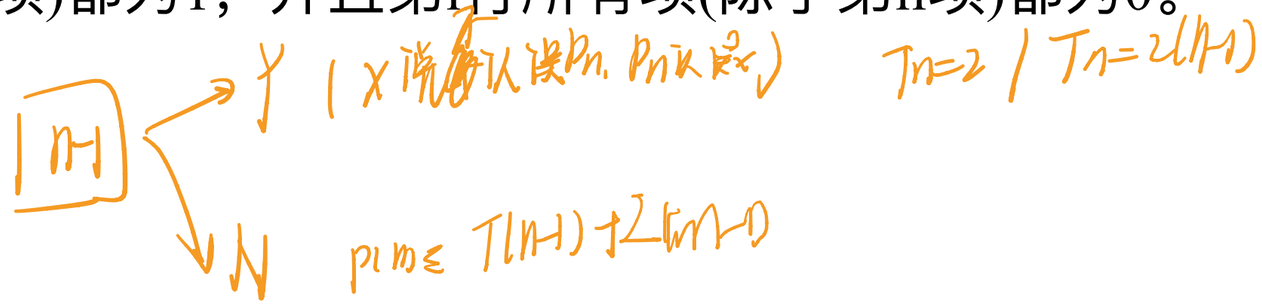
总共的步骤数是O(n)

社会名流问题

在n个人中，一个被所有人知道但却不知道别人的人，被定义为社会名流。

最坏情况下可能需要问n(n-1)个问题（你认识某人吗？）

问题：给定一个n×n邻接矩阵，确定是否存在一个i，其满足在第i列所有项(除了第ii项)都为1，并且第i行所有项(除了第ii项)都为0。



社会名流问题

考察n-1个人和n个人问题的不同。由归纳法，我们假定能够在n-1个人中找到社会名流。由于至多只有一个社会名流，所以有三种可能：(1) 社会名流在最初的n-1人中，(2) 社会名流是第n个人，(3) 没有社会名流。

仍有可能需要n(n-1)次提问！

社会名流问题

考察n-1个人和n个人问题的不同。由归纳法，我们假定能够在n-1个人中找到社会名流。由于至多只有一个社会名流，所以有三种可能：(1) 社会名流在最初的n-1人中，(2) 社会名流是第n个人，(3) 没有社会名流。

仍有可能需要n(n-1)次提问！

“倒推”考虑问题。确定一个社会名流可能很难，但是确定某人不是社会名流可能会容易些。如果我们把某人排除在考虑之外，则问题规模从n减小到n-1。

算法如下：问A是否知道B，并根据答案删除A或者B。假定删除的是A。则由归纳法在剩下的n-1个人中找到一个社会名流。如果没有社会名流，算法就终止；否则，我们检测A是否知道此社会名流，而此社会名流是否不知道A。

```
Algorithm Celebrity(Know)
Input: Know (an n*n Boolean matrix)
Output: celebrity
begin
   i=1;
   j=2;
   next=3;
   {in the first phase we eliminate all but one candidate}
   while next<=n+1 do
      if Know[i,j] then i=next
      else j=next;
      next=next+1;
   if i=n+1 then candidate=j
   else candidate=i
   {now we check that the candidate is indeed the celebrity}
   wrong:=false;
   k=1;
   Know[candidate,candidate]=false;
   while not wrong and k<=n do
      if Know[candidate,k] then wrong=true;
      if not Know[k,candidate] then
         if candidate!=k then wrong=true;
      k=k+1;
   if not wrong then celebrity=candidate
   else celebrity=0
end
```

算法被分为两个阶段：
1）通过消除只留下一个候选者，
2）检查这个候选者是否就是社会名流。
至多要询问3(n-1)个问题：
第一阶段的n-1个问题用于消除n-1个人，
而为了验证侯选者就是社会名流至多要2(n-1)个问题。

轮廓问题----分治算法

问题：给定城市里几座矩形建筑的外形和位置，画出这些建筑的(两维)轮廓，并消去隐藏线。

建筑B_i通过三元组(L_i，H_i，R_i)来表示。L_i和R_i分别表示建筑的左右x坐标，而H_i表示建筑的高度。一个轮廓是一列x坐标以及与它们相连的高度，按照从左到右排列。

直接方法：每次加一个建筑，求出新的轮廓线，但总的步数为O(n^2)

一个分治算法：轮廓问题

分治算法后的关键思想是：在最坏情况下，把一栋建筑与已有轮廓合并只需要线性的时间，并且两个不同轮廓合并也只需要线性的时间。

使用类似于把一栋建筑与已有轮廓合并的算法，就能够把两个轮廓合并。我们从左到右同时扫描两个轮廓，匹配x坐标，并在需要时调整高度。这个合并可以在线性时间内完成，因此在最坏情况下完整的算法运行时间是O(nlogn)。

分治算法

分治法通过减小问题规模的途径来降低求解的难度。

但是分治法通常将问题拆分成多个相互独立但结构与原问题一致的小规模问题，然后将求解得到的小规模问题的解组合起来得到原问题的解。

分治算法框架

divide-and-conquer(P)

1. if |P|<=n_0 then solve(P)
2. 将P分拆为小规模的子问题P_1, P_2, ..., P_k
3. for i=1 to k
4. y_i=divide-and-conquer(P_i)
5. end for
6. return merge(y_1,...,y_k)

从这一算法流程可以看到，分治法可以分为三个部分：

分(divide)：将规模为n的原问题分拆为k≥1个小问题，每个小问题的规模都严格小于n；

治(conquer)：如果小问题的规模大于事先设定的阈值n_0，则递归求解，否则直接求解小规模问题

合(merge)：将k个子问题的解组合形成原问题的解

$$T(n) = div(n) + \sum_{i=1}^{k} T(n_i) + merge(n)$$

在二叉树中计算平衡因子

令T是一个根为r的二叉树。节点v的高度是v和树下方最远叶子的距离。节点v的平衡因子被定义成它的左子树的高度与右子树的高度的差

问题：给定一个n个节点的二叉树T，计算它的所有节点的平衡因子

归纳假设：我们已知如何计算节点数<n的二叉树的全部节点的平衡因子。

然而，根的平衡因子，并不依赖于它的儿子的平衡因子，而是依赖于它们的高度，而高度很容易计算。

更强的归纳假设：已知如何计算节点数<n的二叉树的全部节点的平衡因子和高度。

寻找最大连续子序列

问题：给定实数序列x_1，x_2，…，x_n(不需要是正数)，寻找连续子序列x_i，x_{i+1}，…，x_j，使得其数值之和在所有的连续子序列数值之和中为最大。称这个子序列为最大子序列。

归纳假设：已知如何找到规模<n的序列的最大子序列。

考虑规模n>1的序列S=(x_1，x_2，…，x_n)。由归纳假设已知如何在S/=(x_1，x_2，…，x_{n-1})中找到最大子序列。如果其最大子序列为空，则S/中所有的数值为负数，我们仅需考察x_n。假设通过归纳法在S/中找到的最大子序列是$S/_M$=(x_i，x_{i+1}，…，x_j)，1≤i≤j≤n-1。如果j=n-1(即最大子序列是S/后缀)，则容易把这个解扩展到S中：若X_n是正数，则把它加到$S/_M$中；否则，$S/_M$仍是最大子序列。如果j<n-1，则或者$S/_M$仍是最大，或者存在另一个子序列，它在S/中不是最大，但在增加了x_n的S中是最大者。

寻找最大连续子序列

更强的归纳假设：已知如何找到规模<n的序列的最大子序列，以及作为后缀的最大子序列。

如果知道这两个子序列，算法就明确了。我们把x_n加到最大后缀中，如果它的和大于原来的最大子序列，则得到一个新的最大子序列(同样也是一个后缀)，否则，保留以前的最大子序列。但求解过程还没有完全结束，还需要寻找新的最大后缀子序列。我们不能总是简单地把x_n加到以前的最大后缀中，有可能以x_n结束的最大后缀的和是负数，在此情况下，把空集作为最大后缀。

最大连续子序列算法

```
Algorithm Maximum_Consecutive_Subsequence(X,n)
Input: X (an array of size n)
Output: Global_Max (the sum of the maximum subsequence)
begin
   Global_Max:=0;
   Suffix_Max:=0;
   for i:=1 to n do
      if x[i]+Suffix_Max>Global_max then
         Suffix_Max:=Suffix_Max+x[i];
         Global_Max:=Suffix_Max
      else  if x[i]+Suffix_Max>0 then
               Suffix_Max:=x[i]+Suffix_Max
         else Suffix_Max:=0
end
```

增强归纳假设

当试图用归纳方式证明时，我们经常遇到以下情节：用P来表示定理，归纳假设可以用P(<n)表示，而证明必须推导出P(n)，即P(<n)⇒P(n)。在许多情况下，我们可以增加另一个假设，称之为Q，从而使证明变得容易，即证明[P and Q](<n)⇒P(n)比证明P(<n)⇒P(n)容易

在使用这个技巧时，人们最易犯的错误是增加的额外假设本身必须也有相应的证明。换句话说，当他们证明[P and Q](<n)⇒P(n)时，忘记了Q是假定的。

至关重要的是要精确地按照归纳假设进行问题求解。

背包问题----动态规划

问题：给定一个整数K和n个不同大小的物品，第i个物品的大小为整数k_i，寻找一个物品的子集，它们的大小之和正好为K，或者确定不存在这样的子集。

用P(n，K)表示该问题，其中n表示物品的数目而K表示背包的大小。关注判定问题。

归纳假设(最初的设想)：已知如何求解P(n-1，K)。但如果设对于P(n-1，K)不存在解。我们可以使用这个否定的结论吗?

归纳假设(第二次设想)：我们已知如何求解P(n-1，k)，其中0≤k≤K。但这个算法可能是低效率的，我们把一个规模为n的问题归约到了两个规模为n-1的子问题!

背包问题----动态规划

关键：全部的可能问题数目不是很大，在许多次得到的是同样的问题。可以在求解中记住已有的解答，从而对相同的问题不作第二次求解。

方法：把增强的归纳假设与强归纳(它不仅利用n-1时的解，而是利用所有较小规模情形时的解)结合起来。

动态规划的本质是把所有前面已知的结果建成一个大表格。这个表格是被迭代构造的。每一项是由矩阵中它上面的其它项结合计算得出，或者是由它左边的项结合计算得出。主要的问题是用最高效的方式来组织矩阵的构造。

由表格前面已行算好的项求解.

关键 → 怎样组织表格.

```
Algorithm Knapsack(S,K)
Input: S (an array of size n storing the sizes of the items), K (the size of the knapsack)
Output: P (a 2D-array such that P[i,k].exist=true if there exists a solution with the first i elements
     and a knapsack of size k, and P[i,k].belong=true if the ith element belongs to that solution
begin
   P[0,0].exist:=true;
   for k:=1 to K do
      P[0,k].exist:=false;
   for i:=1 to n do
      for k:=0 to K do
         P[i,k].exist:=false;
         if P[i-1,k].exist then
            P[i,k].exist:=true;
            P[i,k].belong:=false;
         else if k-S[i]>=0 then
            if P[i-1,k-S[i]].exist then
               P[i,k].exist:=true;
               P[i,k].belong:=true;
end
```

求P[i,j]

↓

P[i-1,j], P[i-1,j-ki]

复杂性：在表格中有nK个项，每一项由其他两项在常数式时间内计算所得。因此，总共的运行时间是O(nK)。

第i个物体可以放可能

剩余空间能否用前i-1个物品填满

动态规划

如果算法在求解问题的过程中需要反复计算某些子问题，则可以从小规模问题开始求解，并将其解存储起来，在求解大规模问题时，如果需要某些子问题的解，则利用存储内容直接获得其解，避免重复计算。

与分治法的差异

子问题不独立：动态规划

子问题独立：递归

常见错误

与已经讨论的归纳证明的常见错误类似。例如，忘记了基础情形是比较常见的。在递归过程中，基础情形对于终止递归是必需的。另一常见错误是，把对于n的解扩展到问题对于n+1时的一个特定实例的解，而不是对于任意实例。无意识的改变假设是另一个常见错误。

小结

通过把问题的一个实例归约到一个或多个较小规模的实例，可以使用归纳原理来设计算法。如果归约总是能实现，并且基础情形可以被解决，则算法可通过归纳进行设计。因此，主要的思想是如何归约问题，而不是直接对问题求解。

把问题规模减小的一种最容易的方法是去除问题中的某些元素。这个技术应该是处理问题首要手段，可以有许多不同的方式，除了简单地去除元素外，把两个元素合并为一个也是可能的，或者找到一个在特定(容易)情况下可以处理的元素，或者引入一个新元素来取代原来的两个或几个元素。

可以用多种方式来归约问题的规模。然而，不是所有的归约都有同样的效率，因此要考虑所有归约的可能性，特别是考虑不同的归纳次序。

小结

减小问题规模的一个最有效的方法是把它分成两个(或多个)相等规模的部分。如果问题可以被分割，则分治法非常有效，其中子问题的输出可以容易地生成全问题的输出。

由于归约只能改变问题的规模，并不改变问题本身，所以应该寻找尽可能独立的小规模的子问题。

有一种方法来克服归约问题必须与原始问题一致的局限：改变问题的描述。这是一个经常使用的非常重要的方法，有时，它比削弱假设要好，可获得一个较弱的算法并作为完整算法中的一个步骤来使用。

这些技术可以同时一起使用，或者作不同的组合。

归纳法（尾递归），分治法，动态规划