**http://blog.csdn.net/hguisu/article/details/7709276**

## 回溯是啥

用爬山来比喻回溯，好比从山脚下找一条爬上山顶的路,起初有好几条道可走,当选择一条道走到某处时,又有几条岔道可供选择,只能选择其中一条道往前走,若能这样子顺利爬上山顶则罢了,否则走到一条绝路上时,只好返回到最近的一个路口,重新选择另一条没走过的道往前走。如果该路口的所有路都走不通,只得从该路口继续回返。照此规则走下去,要么找到一条到达山顶的路,要么最终试过所有可能的道,无法到达山顶。  
回溯是一种穷举，但与brute force有一些区别，回溯带了两点脑子的，并不多，brute force一点也没带。  
第一点脑子是回溯知道回头；相反如果是brute force,发现走不通立刻跳下山摔死，换第二条命从头换一条路走。  
第二点脑子是回溯知道剪枝；如果有一条岔路上放了一坨屎，那这条路我们不走，就可以少走很多不必要走的路。

还有一些爱混淆的概念：递归，回溯，DFS。  
回溯是一种找路方法，搜索的时候走不通就回头换路接着走，直到走通了或者发现此山根本不通。  
DFS是一种开路策略，就是一条道先走到头，再往回走一步换一条路走到头，这也是回溯用到的策略。在树和图上回溯时人们叫它DFS。  
递归是一种行为，回溯和递归如出一辙，都是一言不合就回到来时的路，所以一般回溯用递归实现；当然也可以不用，用栈。  
以下以回溯统称，因为这个词听上去很文雅。

定义：

        回溯法(探索与回溯法)是一种选优搜索法，按选优条件向前搜索，以达到目标。但当探索到某一步时，发现原先选择并不优或达不到目标，就退回一步重新选择，这种走不通就退回再走的技术为回溯法，而满足回溯条件的某个状态的点称为“回溯点”。

## 识别回溯

判断回溯很简单，拿到一个问题，你感觉如果不穷举一下就没法知道答案，那就可以开始回溯了。  
一般回溯的问题有三种：

1. Find a path to success 有没有解
2. Find all paths to success 求所有解
   * 求所有解的个数
   * 求所有解的具体信息
3. Find the best path to success 求最优解

理解回溯：给一堆选择, 必须从里面选一个. 选完之后我又有了新的一组选择. This procedure is repeated over and over until you reach a final state. If you made a good sequence of choices, your final state is a goal state; if you didn't, it isn't.

回溯可以抽象为一棵树，我们的目标可以是找这个树有没有good leaf，也可以是问有多少个good leaf，也可以是找这些good leaf都在哪，也可以问哪个good leaf最好，分别对应上面所说回溯的问题分类。good leaf都在leaf上。good leaf是我们的goal state，leaf node是final state，是解空间的边界。

2、有组织的穷举式搜索：回溯法的基本做法是搜索或者有的组织穷尽搜索。它能避免搜索所有的可能性。即避免不必要的搜索。这种方法适用于解一些组合数相当大的问题。

3、搜索解空间树：回溯法在问题的解空间树中，按DFS，从根结点出发搜索解空间树。**[算法](http://lib.csdn.net/base/datastructure" \o "算法与数据结构知识库" \t "_blank)**搜索至解空间树的任意一点时，先判断该结点是否包含问题的解。如果肯定不包含（剪枝过程），则跳过对该结点为根的子树的搜索，逐层向其祖先结点回溯；否则，进入该子树，继续按深度优先策略搜索。

## 2. 问题的解空间 和空间树

        这个空间必须至少包含一个解（可能是最优的）。 一个复杂问题的解决往往由多部分构成，即，一个大的解决方案可以看作是由若干个小的决策组成。很多时候它们构成一个决策序列。解决一个问题的所有可能的决策序列构成该问题的解空间。解空间中满足约束条件的决策序列称为可行解。 在约束条件下使目标值达到 称为该问题的最优解。在解空间中，前k项决策已经取定的所有决策序列之集称为k定子解空间。0定子解空间即是该问题的解空间。

      问题的解空间通常是在搜索问题的解的过程中动态产生的，这是回溯算法的一个重要特性。

**解空间树**  
       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合树。此时，解空间有http://my.csdn.net/uploads/201206/28/1340851959_9663.jpg个元素，遍历子集树的任何算法均需的http://my.csdn.net/uploads/201206/27/1340790923_4229.jpg计算时间。

如例:定和子集问题(给一个array= {W1,w2, ... Wn}，一个target=M求所有子集合S，其sum=M。

这个问题的解可以表示成0/1数组{x1,x2,…,xn}，依据W1是否属于S， X1分别取值1或0。故解空间中共有http://my.csdn.net/uploads/201206/28/1340851959_9663.jpg个元素。它的树结构是一棵完整二叉树。 由根到所有其它节点的路径描述了这个问题的状态空间。

当所给的问题是确定n个元素的满足某种性质的排列时，**相应的解空间树称为排列树**，此时，解空间有个元素。遍历排列树的任何算法均需的计算时间，均需的http://my.csdn.net/uploads/201206/27/1340790923_4229.jpg计算时间。



回溯法就从初始节点（解空间树的根节点）出发，以深度优先的方式搜索整个解空间

当我们将问题的有关数据以一定的**[数据结构](http://lib.csdn.net/base/datastructure" \o "算法与数据结构知识库" \t "_blank)**存储好以后（例如，旅行商问题存储赋权图的邻接矩阵、定和子集问题是存储已知的n+1个数、4皇后问题用整数对(i,j)表示棋盘上各个位置，不必先建立一个解空间树），就搜索生成解空间树的一部分或全部，并寻找所需要的解。也就是说，对于实际问题不必生成整个状态空间树，然后在整个解空间中搜索，我们只需有选择地搜索。为了使搜索更加有效，常常在搜索过程中加一些判断以决定搜索是否该终止或改变路线。通常采用两种策略：

其一是使用约束函数，在扩展节点处剪去不满足约束的子树；

其二是用限界函数，“剪去”不能达到最优解的子树。

这两种函数统称为剪枝函数。

**总结：**

扩展结点:一个正在产生儿子的结点称为扩展结点

活结点:一个自身已生成但其儿子还没有全部生成的节点称做活结点

死结点:一个所有儿子已经产生的结点称做死结点

深度优先的问题状态生成法：如果对一个扩展结点R，一旦产生了它的一个儿子C，就把C当做新的扩展结点。在完成对子树C（以C为根的子树）的穷尽搜索之后，将R重新变成扩展结点，继续生成R的下一个儿子（如果存在）

宽度优先的问题状态生成法：在一个扩展结点变成死结点之前，它一直是扩展结点。

回溯法：为了避免生成那些不可能产生最佳解的问题状态，要不断地利用限界函数(bounding function)来处死(剪枝)那些实际上不可能产生所需解的活结点，以减少问题的计算量。具有限界函数的深度优先生成法称为回溯法。（回溯法 = 穷举 +　剪枝）。

**基本思路：**

已经有（x1,x2,x3,……xi），开始添加x(i+1)属于s(i+2)，检查 (x1,x2,……,xi,x(i+1))是否满足条件，满足了就继续添加x(i+2)、s(i+2)，若所有的x(i+1)属于s(i+1)都不能得到 部分解，就去掉xi，回溯到(xi,x2,……x(i- 1))，添加那些未考察过的x1属于s1，看其是否满足约束条件，为此反复进行，直至得到解或证明无解

## 4.回溯法的步骤

总结起来，运用回溯法解题通常包括以下三个步骤   
1).确定问题的解空间 ：针对所给问题，定义问题的解空间；

子集树问题：装载问题、符号三角形问题、0-1背包、最大团问题  
排列树问题：批处理作业调度、n-queen、旅行售货员问题、圆排列问题、电路板排列问题  
其他：图的m着色问题

2).确定易于搜索的解空间结构：

找出适当的剪枝函数，约束函数和限界函数。

3).以深度优先的方式搜索解空间，并且在搜索过程中用剪枝函数避免无效的搜索。

递归回溯

迭代回溯

4）利用限界函数避免移动到不可能产生解的子空间

## 5.算法框架

**1. 递归回溯：**

回溯法对解空间作深度优先搜索，因此，在一般情况下用递归方法实现回溯法。

def backtracking(t):

if t > n: #到达叶子结点，将结果输出

output(n)

else: #遍历结点t的所有子结点，即枚举t所有可能的路径. f(n,t)=下界;g(n,t)=上界;

for i in range(f(n,t),g(n,t)):

x[t] = h[i] #

if constraint(t) and bound(t): #满足界限函数和约束函数，则继续遍历，进入下一层

backtrack(t+1)

//回溯前的清理工作（如x[t]=h[i-1]等

else：#如不满足，就剪枝

t是递归深度；  
n是深度控制，即解空间树的的高度；  
可行性判断有两方面的内容：不满约束条件则剪去相应子树；若限界函数越界，也剪去相应子树；两者均满足则进入下一层；

**2. 迭代回溯**

采用树的非递归深度优先遍历算法，可将回溯法表示为一个非递归迭代过程。

// 针对N叉树的迭代回溯方法

iterative\_backtrack()

t = 1

while t > 0: //有路可走

if f(n,g) <= g(n,t):

for i in range(f(n,t),g(n,t)): //  遍历结点t的所有子结点

x[t] = h[i]

if (constraint(t) and bound(t)):// 剪枝

if solution(t):// 找到问题的解，输出结果

output(x)

else:// 未找到，向更深层次遍历

t+=1

else:

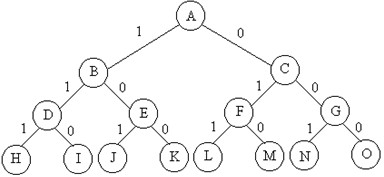
t-=1

## 6. 回溯法依赖的两种数据结构

**回溯法通常在解空间树上进行搜索，一般依赖的两种数据结构：子集树和排列树**

**子集树**（遍历子集树需O(2^n)计算时间）：

一般有装载问题、符号三角形问题、0-1背包问题、最大团问题



def backtracking(t):

if t > 0: // 到达叶子结点

outout(x)

else:

for i in range(0,2):

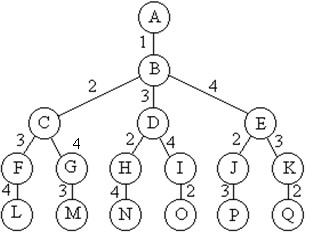
x[t] = i

if legal(t): // 约束函数

backtracking(t+1)

**排列树**（遍历排列树需要O(n!)计算时间）：

一般有批处理作业调度、n后问题、旅行售货员问题、圆排列问题、电路板排列问题

****

def backtracking(t):

if t > 0: // 到达叶子结点

outout(x)

else:

for i in range(t,n+1):// 完成全排列

swap(x[t],x[i])

if legal(t): // 约束函数

backtracking(t+1)

swap(x[t],x[i])

其中

f(n,t),g(n,t)表示当前扩展结点处未搜索过的子树的起始标号和终止标号,

h(i)表示当前扩展节点处，

x[t]第i个可选值

constraint(t)和bound(t)是当前 扩展结点处的约束函数和限界函数。constraint(t)返回true时，在当前扩展结点 x[1:t]取值满足约束条件，否则不满足约束条件，可减去相应的子树。bound(t)返 回的值为true时，在当前扩展结点x[1:x]处取值未使目标函数越界，还需要由backtrack(t+1) 对其相应的子树进一步搜索。

**关于回溯的三种问题，模板略有不同，**第一种，返回值是true/false。  
第二种，求个数，设全局counter，返回值是void；求所有解信息，设result，返回值void。  
第三种，设个全局变量best，返回值是void。

第一种：

boolean solve(Node n) {

**if** n **is** a leaf node {

**if** the leaf **is** a goal node, **return** true

**else** **return** false

} **else** {

**for** each child c of n {

**if** solve(c) succeeds, **return** true

}

**return** false

}

}

第二种：

void solve(**Node** **n**) {

if n is a leaf **node** **{**

**if** the leaf is a goal **node, count**++, return;

else return

} else {

for each child c of n {

solve(c)

}

}

}

第三种：

void solve(**Node** **n**) {

if n is a leaf **node** **{**

**if** the leaf is a goal **node, update** best result, return;

else return

} else {

for each child c of n {

solve(c)

}

}

}

## 7.回溯法的应用

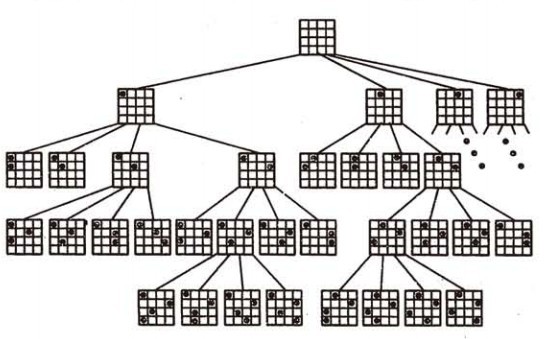
应用回溯法有：

* 1）装载问题
* 2）批处理作业调度
* 3）符号三角形问题
* 4）n后问题
* 5）0-1背包问题
* 6）最大团问题
* 7）图的m着色问题
* 8）旅行售货员问题
* 9）圆排列问题
* 10）电路板排列问题
* 11）连续邮资问题

### **7.1 n皇后问题：**

**1.问题表述：**在n×n格的棋盘上放置彼此不受攻击的n个皇后。按照国际象棋的规则，皇后可以攻击与之处在同一行或同一列或同一斜线上的棋子。n后问题等价于在n×n格的棋盘上放置n个皇后，**任何2个皇后不放在同一行或同一列或同一斜线上**。求不同的解的个数。

复杂问题从简单问题入手，我们先分析四皇后的问题，四叉树展示了求解的过程：



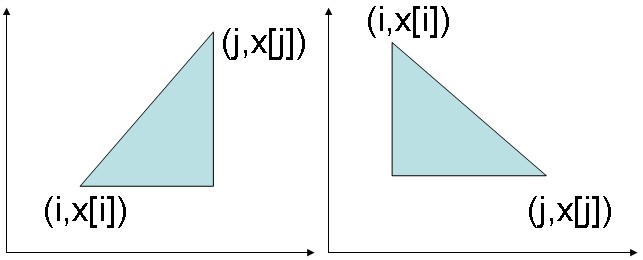
**2. 问题分析**：

(1) 解空间：一组n元一维向量(x1, x2, x3, ... , xn),搜索空间是：1<=xi<=n， i=1,2,3,...,n

(2) 约束条件：

1)不同列：xi != xj

2)不处于同一正、反对角线：|i-j| != |x(i)-x(j)|



**3. 代码实现：**

1. // stdafx.h : include file for standard system include files,
2. // or project specific include files that are used frequently, but
3. // are changed infrequently
4. //
6. #pragma once
8. #include <stdio.h>
9. #include "stdlib.h"
10. #include <iostream>
11. **using** **namespace** std;

14. //宏定义
15. #define TRUE   1
16. #define FALSE   0
17. #define OK    1
18. #define ERROR   0
19. #define INFEASIBLE -1
20. #define OVERFLOW -2
22. **typedef** **int** Status;
23. **typedef** **int** ElemType;

Test.cpp

1. // Test.cpp : Defines the entry point for the console application.
2. //
3. #include "stdafx.h"
5. #include <vector>

8. **class** queen
9. {
10. // 皇后在棋盘上的位置
11. **struct** q\_place {
12. **int** x;
13. **int** y;
14. q\_place ()
15. : x(0),y(0)
16. {}
17. };

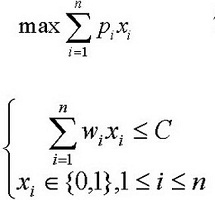

21. **public**:
22. queen(**int** qc)
23. : q\_count (qc), sum\_solution (0) {
24. curr\_solution.resize (q\_count);
25. }
27. **void** backtrack () {
28. \_backtracking (0);
29. }
31. **private**:
32. /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/
33. /\*  判断对应的位置是否存在当前的方案中
34. \*/
35. /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/
36. **bool** \_isCoordinate(**int** x, **int** y)
37. {
38. **for** (**size\_t** i = 0;i < curr\_solution.size(); ++ i) {
39. **if** (curr\_solution[i].x ==x &&  curr\_solution[i].y == y) {
40. **return** **true**;
41. }
42. }
43. **return** **false**;
44. }
45. /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/
46. /\*  打印当前的位置
47. \*/
48. /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/
49. **void** \_printResult()
50. {
51. **for** (**size\_t** i = 0;i < curr\_solution.size(); ++ i) {
52. **for**(**size\_t** j = 0;j < curr\_solution.size(); ++j) {
53. **if** (\_isCoordinate(i, j)) {
54. cout<<"1 ";
55. }**else**{
56. cout<<"0 ";
57. }
58. }
59. cout<< endl;
61. }
62. cout << "sum\_solution = " << sum\_solution << endl;
63. }
65. /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/
66. /\*  现在从第i行算起继续为后续的棋子选择合适的位置
67. \*/
68. /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/
69. **void** \_backtracking (**int** i)
70. {
71. **if** (i >= q\_count) { //找到一个解决方案，将结果输出
72. ++ sum\_solution ;
73. \_printResult();
74. }
75. **else** {
76. **for** (**int** j = 0;j < q\_count; ++ j) {
77. //将第i行第j列放置一个棋子
78. curr\_solution[i].x = j;
79. curr\_solution[i].y = i;
80. **if** (isOk(i)) { //当前布局合法
81. \_backtracking (i + 1);
82. }
83. }
84. }
85. }
87. /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/
88. /\*  判断第k个皇后的位置是否与前面的皇后相冲突
89. \*/
90. /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/
91. **bool** isOk(**int** k)
92. {
93. **for** (**int** i = 0; i < k; ++ i) {
94. **if** ((abs(curr\_solution[i].x - curr\_solution[k].x) == abs(curr\_solution[i].y - curr\_solution[k].y))
95. || curr\_solution[i].x == curr\_solution[k].x) {
96. **return** **false**;
97. }
98. }
99. **return** **true**;
100. }
102. **private**:
103. vector<q\_place> curr\_solution;    // 当前解决方案
104. **const** **int** q\_count;              // 皇后个数
105. **int** sum\_solution;               // 当前找到的解决方案的个数
106. };


110. **int** main()
111. {
112. queen q(5);
113. q.backtrack ();
114. **return** 0;
115. }

### **7.2 定和0/1背包问题**

问题表述：给定n种物品和一背包。第i件物品的重量是wi，其价值为pi，背包的容量为C。问应如何选择装入背包的物品，使得装入背包中物品的总价值最大?

0-1背包问题是一个数规划问题：确定一个向量：x=（x1,x2,...,xn）满足：



例如：n=3但是时候：

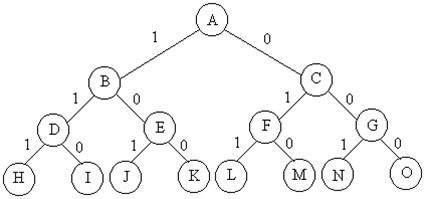
W = (10, 8,5)

p = (5,5,1)

C = 16;

最优解为：(1,01),此时价值为：6

**0/1背包问题用完全二叉树表示的解空间：**



问题分析：

(1) 解空间：一组n元一维向量(x1, x2, x3, ... , xn),搜索空间是：1<=xi<=n， i=1,2,3,...,n

(2) 约束条件：

**可行性约束函数：**

**http://my.csdn.net/uploads/201206/29/1340953369_6769.jpg**

**上界函数：**

考虑一个右子树的时候，设

r：是当前未考虑的剩余物品的总价值(remainder)

cp：是当前的价值(current price)

bestp：是当前得到的最优价值(best price)

http://my.csdn.net/uploads/201206/29/1340951369_8802.jpg

那么，满足：

但是，上界r太松。

一个更加紧的上界：

将剩余物品按照单位重量价值排序，然后依次装入物品，直到装不下，再将剩余物品的一部分放入背包。(r\_n  <=  r)

[**C语言**](http://lib.csdn.net/base/c)实现：

1. // TestWin32.cpp : Defines the entry point for the console application.
2. //
4. #include "stdafx.h"

7. **int** curr\_weight = 0;        //当前重量
8. **int** curr\_value = 0;     //当前价值
9. **int** bestv = 0;          //最优解
10. **int** x\_length = 0;           //
11. /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/
12. /\*  将物品按单位价格降序排序                                                \*/
13. /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/
14. **void** sortItem(itemGoods \*item, **int** n){
15. itemGoods temp;
16. **for**(**int** i = 0; i < n-1; ++i){
17. **for**(**int** j = i+1; j < n; ++j){
18. **if**((item[i].v/item[i].w) < (item[j].v/item[j].w)){
19. temp = item[i];
20. item[i] = item[j];
21. item[j] = temp;
22. }
23. }
24. }
25. }
26. /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/
27. /\*  边界函数 ： 计算上界
28. @int C, 背包容量
29. @int i     第i个物品
30. @int n     物品个数
31. \*/
32. /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/
33. **int** bound(itemGoods \*item, **int** capacity, **int** i, **int** n){
34. **int** capacity\_left = capacity - curr\_weight;
35. **int** value\_left = curr\_value;
36. // 按物品单位价值递减序装入物品
37. **while**(i <= n && item[i].w <= capacity\_left){
38. capacity\_left -= item[i].w;
39. value\_left += item[i].v;
40. ++i;
41. }
42. //装满背包
43. **if**(i <= n)
44. value\_left += item[i].v \* capacity\_left / item[i].w;
45. **return** value\_left;
46. }
47. /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/
48. /\* 递归回溯
49. @int capacity,  背包容量
50. @int i              第i个物品
51. @int n              物品个数
52. \*/
53. /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/
54. **void** backtrack(itemGoods \*item, **int** capacity, **int** i, **int** n, **int** \*bestX){
55. **if**(i >= n){
56. //到达叶子结点，更新最优价值
57. **if**(bestv < curr\_value){
58. bestv = curr\_value;
59. x\_length = 0;
60. **for**(**int** i = 0; i < n; ++i)
61. **if**(item[i].visited){
62. bestX[x\_length] = item[i].id;
63. ++x\_length;
64. }
65. }
66. **return**;
67. }
68. //搜索左子树：左剪枝，能放的下的物品
69. **if**(curr\_weight + item[i].w <= capacity){
70. curr\_weight += item[i].w;
71. curr\_value += item[i].v;
72. item[i].visited = **true**;
73. backtrack(item,capacity,i+1,n,bestX);
74. curr\_weight -= item[i].w;
75. curr\_value -= item[i].v;
77. }
78. //搜索右子树：放不下的物品
79. **if**(bound(item,capacity,i,n) > bestv)
80. item[i].visited = **false**;
81. backtrack(item,capacity,i+1,n,bestX);
82. }
83. **int** Knapsack(itemGoods \*item, **int** n, **int** capacity, **int** \*bestX){
84. sortItem(item,n);
85. backtrack(item,capacity,0,n,bestX);
86. **return** bestv;
87. }
88. **void** initGoods(itemGoods \*item,**int** n){
90. cout << "物品信息：" << endl;
91. **for**(**int** i = 0; i < n; ++i){
92. item[i].id = i;
93. item[i].visited = **false**;
94. cout << "物品" <<i<<"重量：";
95. cin >> item[i].w;
96. cout << "物品" <<i<<"价值：";
97. cin >> item[i].v;
98. }
100. }
101. **void** printKnapsack(**int** \*bestX, **int** max\_value){
102. cout << "背包的物品id:" << endl;
103. **for**(**int** i = 0; i < x\_length; ++i)
104. cout << bestX[i]+1 << "\t";
105. cout << endl;
106. cout << "最大价值： " << max\_value << endl;
108. }
109. **int** main(){
110. **int** n;
111. cout << "物品数量:" << endl;
112. cin >> n;
113. **int** capacity;
114. cout << "背包容量:" << endl;
115. cin >> capacity;
116. itemGoods \*item = **new** itemGoods[n];
117. initGoods(item, n);
118. **int** \*bestX = **new** **int**[n];  //当前最优解
119. **int** max\_value = Knapsack(item,n,capacity, bestX);
120. printKnapsack(bestX, max\_value);
121. **return** 0;
122. }

c++实现：

1. // stdafx.h : include file for standard system include files,
2. // or project specific include files that are used frequently, but
3. // are changed infrequently
4. //
6. #pragma once
8. #include "targetver.h"
9. #include <stdio.h>
10. #include "stdlib.h"
11. #include <iostream>
12. **using** **namespace** std;
14. //宏定义
15. #define TRUE   1
16. #define FALSE   0
17. #define OK    1
18. #define ERROR   0
19. #define INFEASIBLE -1
20. #define OVERFLOW -2
22. **typedef** **int** Status   ;
23. **typedef** **int** ElemType ;
25. **typedef** **struct** itemGoods{
26. **int** id;
27. **bool** visited;
28. **int** w;
29. **int** v;
30. }itemGoods ;


34. **class** knapsack{
35. **private**:
36. itemGoods \*item ;
37. **int** capacity;   //背包容量
38. **int** n;          //物品数
39. **int** curr\_weight;//当前重量
40. **int** curr\_value; //当前价值
41. Status bestV;   //当前最优值
42. **int** \*bestX;     //当前最优解
43. **int** x\_length;   //最优解的数量
44. **private**:
46. **void** \_sortItem();
47. **int**  \_bound(**int** i);
48. **void** \_backtrack(**int** i); //递归回溯函数
50. **public**:
51. knapsack (itemGoods \*item, **int** c,**int** n)
52. :capacity(c),   n(n), curr\_value(0), bestV (0), curr\_weight(0),x\_length(0),item(item)
53. {
54. bestX = **new** **int**[n];
55. bestX[0]=0;
56. }
57. **int** backtrack () ;
58. **void** printKnapsack();
59. };

Test.cpp

1. // Test.cpp : Defines the entry point for the console application.
2. //
3. #include "stdafx.h"
5. /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/
6. /\*  边界函数 ： 计算上界
7. \*/
8. /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/
9. **int** knapsack::\_bound(**int** i)
10. {
11. //计算上界
12. **int** cleft = capacity - curr\_weight;
13. **int** value\_left = curr\_value;
14. //以物品单位重量价值递减序装入物品
15. **while**(i < n && item[i].w <= cleft) {
16. cleft         -= item[i].w;
17. value\_left    += item[i].v;
18. i++;
19. }
20. //装满背包
21. **if**(i< n)
22. value\_left += item[i].v/item[i].w \* cleft;
23. **return** value\_left;
25. }
27. /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/
28. /\*    递归回溯
29. \*/
30. /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/
31. **void** knapsack::\_backtrack(**int** i)
32. {
33. **if**(i>=n) {
34. **if**(bestV < curr\_value) {
35. bestV = curr\_value;
36. x\_length = 0;
37. **for**(**int** j = 0;j < n;j++)
38. **if**(item[j].visited) {
39. bestX[j] = item[j].id;
40. ++x\_length;
41. }
42. }
43. **return**;
44. }
45. **if**(curr\_weight + item[i].w <= capacity)  {  //搜索左子树
46. item[i].visited = TRUE;
47. curr\_weight += item[i].w;
48. curr\_value  += item[i].v;
49. \_backtrack(i+1);
50. curr\_weight -= item[i].w;
51. curr\_value  -= item[i].v;
52. }
53. **if**(\_bound(i+1)>bestV) { //搜索右子树
54. item[i].visited = FALSE;
55. \_backtrack(i+1);
56. }
57. }

60. /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/
61. /\*  排序  :将物品按单位价格降序排序
62. \*/
63. /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/
64. **void** knapsack::\_sortItem(){
65. itemGoods temp;
66. **for**(**int** i = 0; i < n-1; ++i){
67. **for**(**int** j = i+1; j < n; ++j){
68. **if**((item[i].v/item[i].w) < (item[j].v/item[j].w)){
69. temp = item[i];
70. item[i] = item[j];
71. item[j] = temp;
72. }
73. }
74. }
75. }
76. **int** knapsack::backtrack () {
77. \_sortItem();
78. \_backtrack(0);
79. **return**  bestV;
80. }
82. **void** knapsack::printKnapsack(){
83. cout << "背包的物品id:" << endl;
84. **for**(**int** i = 0; i < x\_length; ++i)
85. cout << bestX[ i] << "\t";
86. cout << endl;
87. cout << "最大价值： " << bestV << endl;
89. }


93. **int** main(){
94. **int** n = 3;
95. cout << "物品数量:" << endl;
96. //cin >> n;
97. **int** capacity = 5;
98. cout << "背包容量:" << endl;
99. //cin >> capacity;
100. itemGoods \*item = **new** itemGoods[n];
101. //初始化物品
102. //cout << "物品信息：" << endl;
103. //for(int i = 0; i < n; ++i){
104. //  item[i].id = i;
105. //  item[i].visited = FALSE;
106. //  cout << "物品" <<i<<"重量：";
107. //  cin >> item[i].w;
108. //  cout << "物品" <<i<<"价值：";
109. //  cin >> item[i].v;
110. //}
111. item[0].id = 0;
112. item[0].visited = FALSE;
113. item[0].w =2;
114. item[0].v = 2;
116. item[1].id = 1;
117. item[1].visited = FALSE;
118. item[1].w = 2;
119. item[1].v = 2;

122. item[2].id = 2;
123. item[2].visited = FALSE;
124. item[2].w  =4;
125. item[2].v = 10;
127. knapsack ks(item,capacity,n);
128. **int** max\_value = ks.backtrack();
129. ks.printKnapsack();
130. **return** 0;
132. }

### **7.3 素数环问题**

将从1到n这n个整数围成一个圆环，若其中任意2个相邻的数字相加，结果均为素数，那么这个环就成为素数环。

要求输出：从整数1开始。

分析问题，可以构造解空间树，比较顺利的想到 DFS 、回溯。

1. def def(cur,n):
2. if cur == n and is\_prime(res[n-1] + res[0]): /别忘了测试边界，最后一个和第一个数 构成的环
3. for i in range(0,n-1):
4. print res[i]
5. print res[i]
6. else:
7. for i in range(2,n+1):// 尝试每个i, 1始终在排头,因此从2开始计算
8. if vis[i] and is\_prime(res[cur-1]+i):// i未用过且和前一个数和为素数
9. res[cur] = i
10. vis[i] = 1 // 设置标志
11. dfs(cur+1,n)
12. vis[i] = 0// 回溯, 清除标识
13. def is\_prime(m):
14. for i in range(0,m):
15. ok = 1
16. for j in range(2,j\*j<i+1):
17. if i%j == 0:
18. ok = 0
19. break
20. if ok:
21. return i