**Unity3D教程：游戏开发算法（二）**

Posted on 2013年06月25日 by U3d / [Unity3D 基础教程](http://www.unitymanual.com/category/manual/unity3d-%e5%9f%ba%e7%a1%80%e6%95%99%e7%a8%8b)/被围观 19 次

四、递归

递归是设计和描述算法的一种有力的工具，由于它在复杂算法的描述中被经常采用，为此在进一步介绍其他算法设计方法之前先讨论它。

能采用递归描述的算法通常有这样的特征：为求解规模为N的问题，设法将它分解成规模较小的问题，然后从这些小问题的解方便地构造出大问题的解，并且这些规模较小的问题也能采用同样的分解和综合方法，分解成规模更小的问题，并从这些更小问题的解构造出规模较大问题的解。特别地，当规模N=1时，能直接得解。

【问题】 编写计算斐波那契（Fibonacci）数列的第n项函数fib（n）。

斐波那契数列为：0、1、1、2、3、……，即：

|  |  |  |
| --- | --- | --- |
|  |  |  |

|  |  |
| --- | --- |
| 1 | fib(0)=0; |
| 2 |  |
| 3 | fib(1)=1; |
| 4 |  |
| 5 | fib(n)=fib(n-1)+fib(n-2) （当n>1时）。 |

写成递归函数有：

|  |  |  |
| --- | --- | --- |
|  |  |  |

|  |  |
| --- | --- |
| 1 | **int** fib(**int** n) |
| 2 |  |
| 3 | { **if** (n==0) **return** 0; |
| 4 |  |
| 5 | **if** (n==1) **return** 1; |
| 6 |  |
| 7 | **if** (n>1) **return** fib(n-1)+fib(n-2); |
| 8 |  |
| 9 | } |

递归算法的执行过程分递推和回归两个阶段。在递推阶段，把较复杂的问题（规模为n）的求解推到比原问题简单一些的问题（规模小于n）的求解。例如上例中，求解fib(n)，把它推到求解fib(n-1)和fib(n-2)。也就是说，为计算fib(n)，必须先计算fib(n-1)和fib(n-2)，而计算fib(n-1)和fib(n-2)，又必须先计算fib(n-3)和fib(n-4)。依次类推，直至计算fib(1)和fib(0)，分别能立即得到结果1和0。在递推阶段，必须要有终止递归的情况。例如在函数fib中，当n为1和0的情况。

在回归阶段，当获得最简单情况的解后，逐级返回，依次得到稍复杂问题的解，例如得到fib(1)和fib(0)后，返回得到fib(2)的结果，在得到了fib(n-1)和fib(n-2)的结果后，返回得到fib(n)的结果。

在编写递归函数时要注意，函数中的局部变量和参数知识局限于当前调用层，当递推进入“简单问题”层时，原来层次上的参数和局部变量便被隐蔽起来。在一系列“简单问题”层，它们各有自己的参数和局部变量。

由于递归引起一系列的函数调用，并且可能会有一系列的重复计算，递归算法的执行效率相对较低。当某个递归算法能较方便地转换成递推算法时，通常按递推算法编写程序。例如上例计算斐波那契数列的第n项的函数fib(n)应采用递推算法，即从斐波那契数列的前两项出发，逐次由前两项计算出下一项，直至计算出要求的第n项。

【问题】 组合问题

问题描述：找出从自然数1、2、……、n中任取r个数的所有组合。例如n=5，r=3的所有组合为： （1）5、4、3 （2）5、4、2 （3）5、4、1

（4）5、3、2 （5）5、3、1 （6）5、2、1

（7）4、3、2 （8）4、3、1 （9）4、2、1

（10）3、2、1

分析所列的10个组合，可以采用这样的递归思想来考虑求组合函数的算法。设函数为void comb(int m,int k)为找出从自然数1、2、……、m中任取k个数的所有组合。当组合的第一个数字选定时，其后的数字是从余下的m-1个数中取k-1数的组合。这就将求m个数中取k个数的组合问题转化成求m-1个数中取k-1个数的组合问题。设函数引入工作数组a[ ]存放求出的组合的数字，约定函数将确定的k个数字组合的第一个数字放在a[k]中，当一个组合求出后，才将a[ ]中的一个组合输出。第一个数可以是m、m-1、……、k，函数将确定组合的第一个数字放入数组后，有两种可能的选择，因还未去顶组合的其余元素，继续递归去确定；或因已确定了组合的全部元素，输出这个组合。细节见以下程序中的函数comb。

【程序】

|  |  |  |
| --- | --- | --- |
|  |  |  |

|  |  |
| --- | --- |
| 01 | # include |
| 02 |  |
| 03 | # define MAXN 100 |
| 04 |  |
| 05 | **int** a[MAXN]; |
| 06 |  |
| 07 | **void** comb(**int** m,**int** k) |
| 08 |  |
| 09 | { **int** i,j; |
| 10 |  |
| 11 | **for** (i=m;i>=k;i--) |
| 12 |  |
| 13 | { a[k]=i; |
| 14 |  |
| 15 | **if** (k>1) |
| 16 |  |
| 17 | comb(i-1,k-1); |
| 18 |  |
| 19 | **else** |
| 20 |  |
| 21 | { **for** (j=a[0];j>0;j--) |
| 22 |  |
| 23 | printf(“%4d”,a[j]); |
| 24 |  |
| 25 | printf(“\n”); |
| 26 |  |
| 27 | } |
| 28 |  |
| 29 | } |
| 30 |  |
| 31 | } |
| 32 |  |
| 33 | **void** main() |
| 34 |  |
| 35 | { a[0]=3; |
| 36 |  |
| 37 | comb(5,3); |
| 38 |  |
| 39 | } |

【问题】 背包问题

问题描述：有不同价值、不同重量的物品n件，求从这n件物品中选取一部分物品的选择方案，使选中物品的总重量不超过指定的限制重量，但选中物品的价值之和最大。

设n件物品的重量分别为w0、w1、…、wn-1，物品的价值分别为v0、v1、…、vn-1。采用递归寻找物品的选择方案。设前面已有了多种选择的方案，并保留了其中总价值最大的方案于数组option[ ]，该方案的总价值存于变量maxv。当前正在考察新方案，其物品选择情况保存于数组cop[ ]。假定当前方案已考虑了前i-1件物品，现在要考虑第i件物品；当前方案已包含的物品的重量之和为tw；至此，若其余物品都选择是可能的话，本方案能达到的总价值的期望值为tv。算法引入tv是当一旦当前方案的总价值的期望值也小于前面方案的总价值maxv时，继续考察当前方案变成无意义的工作，应终止当前方案，立即去考察下一个方案。因为当方案的总价值不比maxv大时，该方案不会被再考察，这同时保证函数后找到的方案一定会比前面的方案更好。

对于第i件物品的选择考虑有两种可能：

（1） 考虑物品i被选择，这种可能性仅当包含它不会超过方案总重量限制时才是可行的。选中后，继续递归去考虑其余物品的选择。

（2） 考虑物品i不被选择，这种可能性仅当不包含物品i也有可能会找到价值更大的方案的情况。

按以上思想写出递归算法如下：

|  |  |  |
| --- | --- | --- |
|  |  |  |

|  |  |
| --- | --- |
| 01 | **try**(物品i，当前选择已达到的重量和，本方案可能达到的总价值tv) |
| 02 |  |
| 03 | { */\*考虑物品i包含在当前方案中的可能性\*/* |
| 04 |  |
| 05 | **if**(包含物品i是可以接受的) |
| 06 |  |
| 07 | { 将物品i包含在当前方案中； |
| 08 |  |
| 09 | **if** (i　　 **try**(i+1,tw+物品i的重量,tv); |
| 10 |  |
| 11 | **else** |
| 12 |  |
| 13 | */\*又一个完整方案，因为它比前面的方案好，以它作为最佳方案\*/* |
| 14 |  |
| 15 | 以当前方案作为临时最佳方案保存; |
| 16 |  |
| 17 | 恢复物品i不包含状态； |
| 18 |  |
| 19 | } |
| 20 |  |
| 21 | */\*考虑物品i不包含在当前方案中的可能性\*/* |
| 22 |  |
| 23 | **if** (不包含物品i仅是可男考虑的) |
| 24 |  |
| 25 | **if** (i　　 **try**(i+1,tw,tv-物品i的价值)； |
| 26 |  |
| 27 | **else** |
| 28 |  |
| 29 | */\*又一个完整方案，因它比前面的方案好，以它作为最佳方案\*/* |
| 30 |  |
| 31 | 以当前方案作为临时最佳方案保存; |
| 32 |  |
| 33 | } |

为了理解上述算法，特举以下实例。设有4件物品，它们的重量和价值见表：

物品 0 1 2 3

重量 5 3 2 1

价值 4 4 3 1

并设限制重量为7。则按以上算法，下图表示找解过程。由图知，一旦找到一个解，算法就进一步找更好的佳。如能判定某个查找分支不会找到更好的解，算法不会在该分支继续查找，而是立即终止该分支，并去考察下一个分支。

按上述算法编写函数和程序如下：

【程序】

|  |  |  |
| --- | --- | --- |
|  |  |  |

|  |  |
| --- | --- |
| 01 | # include |
| 02 |  |
| 03 | # define N 100 |
| 04 |  |
| 05 | **double** limitW,totV,maxV; |
| 06 |  |
| 07 | **int** option[N],cop[N]; |
| 08 |  |
| 09 | **struct** { **double** weight; |
| 10 |  |
| 11 | **double** **value**; |
| 12 |  |
| 13 | }a[N]; |
| 14 |  |
| 15 | **int** n; |
| 16 |  |
| 17 | **void** find(**int** i,**double** tw,**double** tv) |
| 18 |  |
| 19 | { **int** k; |
| 20 |  |
| 21 | */\*考虑物品i包含在当前方案中的可能性\*/* |
| 22 |  |
| 23 | **if** (tw+a[i].weight<=limitW) |
| 24 |  |
| 25 | { cop[i]=1; |
| 26 |  |
| 27 | **if** (i　　 **else** |
| 28 |  |
| 29 | { **for** (k=0;k　　 option[k]=cop[k]; |
| 30 |  |
| 31 | maxv=tv; |
| 32 |  |
| 33 | } |
| 34 |  |
| 35 | cop[i]=0; |
| 36 |  |
| 37 | } |
| 38 |  |
| 39 | */\*考虑物品i不包含在当前方案中的可能性\*/* |
| 40 |  |
| 41 | **if** (tv-a[i].**value**>maxV) |
| 42 |  |
| 43 | **if** (i　　 **else** |
| 44 |  |
| 45 | { **for** (k=0;k　　 option[k]=cop[k]; |
| 46 |  |
| 47 | maxv=tv-a[i].**value**; |
| 48 |  |
| 49 | } |
| 50 |  |
| 51 | } |
| 52 |  |
| 53 | **void** main() |
| 54 |  |
| 55 | { **int** k; |
| 56 |  |
| 57 | **double** w,v; |
| 58 |  |
| 59 | printf(“输入物品种数\n”); |
| 60 |  |
| 61 | scanf((“%d”,&n); |
| 62 |  |
| 63 | printf(“输入各物品的重量和价值\n”); |
| 64 |  |
| 65 | **for** (totv=0.0,k=0;k　　 { scanf(“%1f%1f”,&w,&v); |
| 66 |  |
| 67 | a[k].weight=w; |
| 68 |  |
| 69 | a[k].**value**=v; |
| 70 |  |
| 71 | totV+=V; |
| 72 |  |
| 73 | } |
| 74 |  |
| 75 | printf(“输入限制重量\n”); |
| 76 |  |
| 77 | scanf(“%1f”,&limitV); |
| 78 |  |
| 79 | maxv=0.0; |
| 80 |  |
| 81 | **for** (k=0;k　　 find(0,0.0,totV); |
| 82 |  |
| 83 | **for** (k=0;k　　 **if** (option[k]) printf(“%4d”,k+1); |
| 84 |  |
| 85 | printf(“\n总价值为%.2f\n”,maxv); |
| 86 |  |
| 87 | } |
| 88 |  |

作为对比，下面以同样的解题思想，考虑非递归的程序解。为了提高找解速度，程序不是简单地逐一生成所有候选解，而是从每个物品对候选解的影响来形成值得进一步考虑的候选解，一个候选解是通过依次考察每个物品形成的。对物品i的考察有这样几种情况：当该物品被包含在候选解中依旧满足解的总重量的限制，该物品被包含在候选解中是应该继续考虑的；反之，该物品不应该包括在当前正在形成的候选解中。同样地，仅当物品不被包括在候选解中，还是有可能找到比目前临时最佳解更好的候选解时，才去考虑该物品不被包括在候选解中；反之，该物品不包括在当前候选解中的方案也不应继续考虑。对于任一值得继续考虑的方案，程序就去进一步考虑下一个物品。

【程序】

|  |  |  |
| --- | --- | --- |
|  |  |  |

|  |  |
| --- | --- |
| 001 | # include |
| 002 |  |
| 003 | # define N 100 |
| 004 |  |
| 005 | **double** limitW; |
| 006 |  |
| 007 | **int** cop[N]; |
| 008 |  |
| 009 | **struct** ele { **double** weight; |
| 010 |  |
| 011 | **double** **value**; |
| 012 |  |
| 013 | } a[N]; |
| 014 |  |
| 015 | **int** k,n; |
| 016 |  |
| 017 | **struct** { **int** ; |
| 018 |  |
| 019 | **double** tw; |
| 020 |  |
| 021 | **double** tv; |
| 022 |  |
| 023 | }twv[N]; |
| 024 |  |
| 025 | **void** next(**int** i,**double** tw,**double** tv) |
| 026 |  |
| 027 | { twv[i].=1; |
| 028 |  |
| 029 | twv[i].tw=tw; |
| 030 |  |
| 031 | twv[i].tv=tv; |
| 032 |  |
| 033 | } |
| 034 |  |
| 035 | **double** find(**struct** ele \*a,**int** n) |
| 036 |  |
| 037 | { **int** i,k,f; |
| 038 |  |
| 039 | **double** maxv,tw,tv,totv; |
| 040 |  |
| 041 | maxv=0; |
| 042 |  |
| 043 | **for** (totv=0.0,k=0;k　　 totv+=a[k].**value**; |
| 044 |  |
| 045 | next(0,0.0,totv); |
| 046 |  |
| 047 | i=0; |
| 048 |  |
| 049 | **While** (i>=0) |
| 050 |  |
| 051 | { f=twv[i].; |
| 052 |  |
| 053 | tw=twv[i].tw; |
| 054 |  |
| 055 | tv=twv[i].tv; |
| 056 |  |
| 057 | **switch**(f) |
| 058 |  |
| 059 | { **case** 1: twv[i].++; |
| 060 |  |
| 061 | **if** (tw+a[i].weight<=limitW) |
| 062 |  |
| 063 | **if** (i　　 { next(i+1,tw+a[i].weight,tv); |
| 064 |  |
| 065 | i++; |
| 066 |  |
| 067 | } |
| 068 |  |
| 069 | **else** |
| 070 |  |
| 071 | { maxv=tv; |
| 072 |  |
| 073 | **for** (k=0;k　　 cop[k]=twv[k].!=0; |
| 074 |  |
| 075 | } |
| 076 |  |
| 077 | **break**; |
| 078 |  |
| 079 | **case** 0: i--; |
| 080 |  |
| 081 | **break**; |
| 082 |  |
| 083 | **default**: twv[i].=0; |
| 084 |  |
| 085 | **if** (tv-a[i].**value**>maxv) |
| 086 |  |
| 087 | **if** (i　　 { next(i+1,tw,tv-a[i].**value**); |
| 088 |  |
| 089 | i++; |
| 090 |  |
| 091 | } |
| 092 |  |
| 093 | **else** |
| 094 |  |
| 095 | { maxv=tv-a[i].**value**; |
| 096 |  |
| 097 | **for** (k=0;k　　 cop[k]=twv[k].!=0; |
| 098 |  |
| 099 | } |
| 100 |  |
| 101 | **break**; |
| 102 |  |
| 103 | } |
| 104 |  |
| 105 | } |
| 106 |  |
| 107 | **return** maxv; |
| 108 |  |
| 109 | } |
| 110 |  |
| 111 | **void** main() |
| 112 |  |
| 113 | { **double** maxv; |
| 114 |  |
| 115 | printf(“输入物品种数\n”); |
| 116 |  |
| 117 | scanf((“%d”,&n); |
| 118 |  |
| 119 | printf(“输入限制重量\n”); |
| 120 |  |
| 121 | scanf(“%1f”,&limitW); |
| 122 |  |
| 123 | printf(“输入各物品的重量和价值\n”); |
| 124 |  |
| 125 | **for** (k=0;k　　 scanf(“%1f%1f”,&a[k].weight,&a[k].**value**); |
| 126 |  |
| 127 | maxv=find(a,n); |
| 128 |  |
| 129 | printf(“\n选中的物品为\n”); |
| 130 |  |
| 131 | **for** (k=0;k　　 **if** (option[k]) printf(“%4d”,k+1); |
| 132 |  |
| 133 | printf(“\n总价值为%.2f\n”,maxv); |
| 134 |  |
| 135 | } |
| 136 |  |

五、回溯法

回溯法也称为试探法，该方法首先暂时放弃关于问题规模大小的限制，并将问题的候选解按某种顺序逐一枚举和检验。当发现当前候选解不可能是解时，就选择下一个候选解；倘若当前候选解除了还不满足问题规模要求外，满足所有其他要求时，继续扩大当前候选解的规模，并继续试探。如果当前候选解满足包括问题规模在内的所有要求时，该候选解就是问题的一个解。在回溯法中，放弃当前候选解，寻找下一个候选解的过程称为回溯。扩大当前候选解的规模，以继续试探的过程称为向前试探。

1、回溯法的一般描述

可用回溯法求解的问题P，通常要能表达为：对于已知的由n元组（x1，x2，…，xn）组成的一个状态空间E={（x1，x2，…，xn）∣xi∈Si ，i=1，2，…，n}，给定关于n元组中的一个分量的一个约束集D，要求E中满足D的全部约束条件的所有n元组。其中Si是分量xi的定义域，且 |Si| 有限，i=1，2，…，n。我们称E中满足D的全部约束条件的任一n元组为问题P的一个解。

解问题P的最朴素的方法就是枚举法，即对E中的所有n元组逐一地检测其是否满足D的全部约束，若满足，则为问题P的一个解。但显然，其计算量是相当大的。

我们发现，对于许多问题，所给定的约束集D具有完备性，即i元组（x1，x2，…，xi）满足D中仅涉及到x1，x2，…，xi的所有约束意味着j（jj。因此，对于约束集D具有完备性的问题P，一旦检测断定某个j元组（x1，x2，…，xj）违反D中仅涉及x1，x2，…，xj的一个约束，就可以肯定，以（x1，x2，…，xj）为前缀的任何n元组（x1，x2，…，xj，xj+1，…，xn）都不会是问题P的解，因而就不必去搜索它们、检测它们。回溯法正是针对这类问题，利用这类问题的上述性质而提出来的比枚举法效率更高的算法。

回溯法首先将问题P的n元组的状态空间E表示成一棵高为n的带权有序树T，把在E中求问题P的所有解转化为在T中搜索问题P的所有解。树T类似于检索树，它可以这样构造：

设Si中的元素可排成xi(1) ，xi(2) ，…，xi(mi-1) ，|Si| =mi，i=1，2，…，n。从根开始，让T的第I层的每一个结点都有mi个儿子。这mi个儿子到它们的双亲的边，按从左到右的次序，分别带权xi+1(1) ，xi+1(2) ，…，xi+1(mi) ，i=0，1，2，…，n-1。照这种构造方式，E中的一个n元组（x1，x2，…，xn）对应于T中的一个叶子结点，T的根到这个叶子结点的路径上依次的n条边的权分别为x1，x2，…，xn，反之亦然。另外，对于任意的0≤i≤n-1，E中n元组（x1，x2，…，xn）的一个前缀I元组（x1，x2，…，xi）对应于T中的一个非叶子结点，T的根到这个非叶子结点的路径上依次的I条边的权分别为x1，x2，…，xi，反之亦然。特别，E中的任意一个n元组的空前缀（），对应于T的根。

因而，在E中寻找问题P的一个解等价于在T中搜索一个叶子结点，要求从T的根到该叶子结点的路径上依次的n条边相应带的n个权x1，x2，…，xn满足约束集D的全部约束。在T中搜索所要求的叶子结点，很自然的一种方式是从根出发，按深度优先的策略逐步深入，即依次搜索满足约束条件的前缀1元组（x1i）、前缀2元组（x1，x2）、…，前缀I元组（x1，x2，…，xi），…，直到i=n为止。

在回溯法中，上述引入的树被称为问题P的状态空间树；树T上任意一个结点被称为问题P的状态结点；树T上的任意一个叶子结点被称为问题P的一个解状态结点；树T上满足约束集D的全部约束的任意一个叶子结点被称为问题P的一个回答状态结点，它对应于问题P的一个解。

【问题】 组合问题

问题描述：找出从自然数1、2、……、n中任取r个数的所有组合。

例如n=5，r=3的所有组合为：

（1）1、2、3 （2）1、2、4 （3）1、2、5

（4）1、3、4 （5）1、3、5 （6）1、4、5

（7）2、3、4 （8）2、3、5 （9）2、4、5

（10）3、4、5

则该问题的状态空间为：

E={（x1，x2，x3）∣xi∈S ，i=1，2，3 } 其中：S={1，2，3，4，5}

约束集为： x1　　 显然该约束集具有完备性。

问题的状态空间树T：

2、回溯法的方法

对于具有完备约束集D的一般问题P及其相应的状态空间树T，利用T的层次结构和D的完备性，在T中搜索问题P的所有解的回溯法可以形象地描述为：

从T的根出发，按深度优先的策略，系统地搜索以其为根的子树中可能包含着回答结点的所有状态结点，而跳过对肯定不含回答结点的所有子树的搜索，以提高搜索效率。具体地说，当搜索按深度优先策略到达一个满足D中所有有关约束的状态结点时，即“激活”该状态结点，以便继续往深层搜索；否则跳过对以该状态结点为根的子树的搜索，而一边逐层地向该状态结点的祖先结点回溯，一边“杀死”其儿子结点已被搜索遍的祖先结点，直到遇到其儿子结点未被搜索遍的祖先结点，即转向其未被搜索的一个儿子结点继续搜索。

在搜索过程中，只要所激活的状态结点又满足终结条件，那么它就是回答结点，应该把它输出或保存。由于在回溯法求解问题时，一般要求出问题的所有解，因此在得到回答结点后，同时也要进行回溯，以便得到问题的其他解，直至回溯到T的根且根的所有儿子结点均已被搜索过为止。

例如在组合问题中，从T的根出发深度优先遍历该树。当遍历到结点（1，2）时，虽然它满足约束条件，但还不是回答结点，则应继续深度遍历；当遍历到叶子结点（1，2，5）时，由于它已是一个回答结点，则保存（或输出）该结点，并回溯到其双亲结点，继续深度遍历；当遍历到结点（1，5）时，由于它已是叶子结点，但不满足约束条件，故也需回溯。

3、回溯法的一般流程和技术

在用回溯法求解有关问题的过程中，一般是一边建树，一边遍历该树。在回溯法中我们一般采用非递归方法。下面，我们给出回溯法的非递归算法的一般流程：

在用回溯法求解问题，也即在遍历状态空间树的过程中，如果采用非递归方法，则我们一般要用到栈的数据结构。这时，不仅可以用栈来表示正在遍历的树的结点，而且可以很方便地表示建立孩子结点和回溯过程。

例如在组合问题中，我们用一个一维数组Stack[ ]表示栈。开始栈空，则表示了树的根结点。如果元素1进栈，则表示建立并遍历（1）结点；这时如果元素2进栈，则表示建立并遍历（1，2）结点；元素3再进栈，则表示建立并遍历（1，2，3）结点。这时可以判断它满足所有约束条件，是问题的一个解，输出（或保存）。这时只要栈顶元素（3）出栈，即表示从结点（1，2，3）回溯到结点（1，2）。

【问题】 组合问题

问题描述：找出从自然数1，2，…，n中任取r个数的所有组合。

采用回溯法找问题的解，将找到的组合以从小到大顺序存于a[0]，a[1]，…，a[r-1]中，组合的元素满足以下性质：

（1） a[i+1]>a[i]，后一个数字比前一个大；

（2） a[i]-i<=n-r+1。

按回溯法的思想，找解过程可以叙述如下：

首先放弃组合数个数为r的条件，候选组合从只有一个数字1开始。因该候选解满足除问题规模之外的全部条件，扩大其规模，并使其满足上述条件（1），候选组合改为1，2。继续这一过程，得到候选组合1，2，3。该候选解满足包括问题规模在内的全部条件，因而是一个解。在该解的基础上，选下一个候选解，因a[2]上的3调整为4，以及以后调整为5都满足问题的全部要求，得到解1，2，4和1，2，5。由于对5不能再作调整，就要从a[2]回溯到a[1]，这时，a[1]=2，可以调整为3，并向前试探，得到解1，3，4。重复上述向前试探和向后回溯，直至要从a[0]再回溯时，说明已经找完问题的全部解。按上述思想写成程序如下：

【程序】

|  |  |  |
| --- | --- | --- |
|  |  |  |

|  |  |
| --- | --- |
| 01 | # define MAXN 100 |
| 02 |  |
| 03 | **int** a[MAXN]; |
| 04 |  |
| 05 | **void** comb(**int** m,**int** r) |
| 06 |  |
| 07 | { **int** i,j; |
| 08 |  |
| 09 | i=0; |
| 10 |  |
| 11 | a[i]=1; |
| 12 |  |
| 13 | **do** { |
| 14 |  |
| 15 | **if** (a[i]-i<=m-r+1 |
| 16 |  |
| 17 | { **if** (i==r-1) |
| 18 |  |
| 19 | { **for** (j=0;j　　 printf(“%4d”,a[j]); |
| 20 |  |
| 21 | printf(“\n”); |
| 22 |  |
| 23 | } |
| 24 |  |
| 25 | a[i]++; |
| 26 |  |
| 27 | **continue**; |
| 28 |  |
| 29 | } |
| 30 |  |
| 31 | **else** |
| 32 |  |
| 33 | { **if** (i==0) |
| 34 |  |
| 35 | **return**; |
| 36 |  |
| 37 | a[--i]++; |
| 38 |  |
| 39 | } |
| 40 |  |
| 41 | } **while** (1) |
| 42 |  |
| 43 | } |
| 44 |  |
| 45 | main() |
| 46 |  |
| 47 | { comb(5,3); |
| 48 |  |
| 49 | } |
| 50 |  |

【问题】 填字游戏

问题描述：在3×3个方格的方阵中要填入数字1到N（N≥10）内的某9个数字，每个方格填一个整数，似的所有相邻两个方格内的两个整数之和为质数。试求出所有满足这个要求的各种数字填法。

可用试探发找到问题的解，即从第一个方格开始，为当前方格寻找一个合理的整数填入，并在当前位置正确填入后，为下一方格寻找可填入的合理整数。如不能为当前方格找到一个合理的可填证书，就要回退到前一方格，调整前一方格的填入数。当第九个方格也填入合理的整数后，就找到了一个解，将该解输出，并调整第九个的填入的整数，寻找下一个解。//Unity3D教程手册：www.unitymanual.com

为找到一个满足要求的9个数的填法，从还未填一个数开始，按某种顺序（如从小到大的顺序）每次在当前位置填入一个整数，然后检查当前填入的整数是否能满足要求。在满足要求的情况下，继续用同样的方法为下一方格填入整数。如果最近填入的整数不能满足要求，就改变填入的整数。如对当前方格试尽所有可能的整数，都不能满足要求，就得回退到前一方格，并调整前一方格填入的整数。如此重复执行扩展、检查或调整、检查，直到找到一个满足问题要求的解，将解输出。

回溯法找一个解的算法：

|  |  |  |
| --- | --- | --- |
|  |  |  |

|  |  |
| --- | --- |
| 01 | { **int** m=0,ok=1; |
| 02 |  |
| 03 | **int** n=8; |
| 04 |  |
| 05 | **do**{ |
| 06 |  |
| 07 | **if** (ok) 扩展; |
| 08 |  |
| 09 | **else** 调整; |
| 10 |  |
| 11 | ok=检查前m个整数填放的合理性; |
| 12 |  |
| 13 | } **while** ((!ok||m!=n)&&(m!=0)) |
| 14 |  |
| 15 | **if** (m!=0) 输出解; |
| 16 |  |
| 17 | **else** 输出无解报告； |
| 18 |  |
| 19 | } |
| 20 |  |

如果程序要找全部解，则在将找到的解输出后，应继续调整最后位置上填放的整数，试图去找下一个解。相应的算法如下：

回溯法找全部解的算法：

|  |  |  |
| --- | --- | --- |
|  |  |  |

|  |  |
| --- | --- |
| 01 | { **int** m=0,ok=1; |
| 02 |  |
| 03 | **int** n=8; |
| 04 |  |
| 05 | **do**{ |
| 06 |  |
| 07 | **if** (ok) |
| 08 |  |
| 09 | { **if** (m==n) |
| 10 |  |
| 11 | { 输出解； |
| 12 |  |
| 13 | 调整； |
| 14 |  |
| 15 | } |
| 16 |  |
| 17 | **else** 扩展; |
| 18 |  |
| 19 | } |
| 20 |  |
| 21 | **else** 调整; |
| 22 |  |
| 23 | ok=检查前m个整数填放的合理性; |
| 24 |  |
| 25 | } **while** (m!=0); |
| 26 |  |
| 27 | } |
| 28 |  |

为了确保程序能够终止，调整时必须保证曾被放弃过的填数序列不会再次实验，即要求按某种有许模型生成填数序列。给解的候选者设定一个被检验的顺序，按这个顺序逐一形成候选者并检验。从小到大或从大到小，都是可以采用的方法。如扩展时，先在新位置填入整数1，调整时，找当前候选解中下一个还未被使用过的整数。将上述扩展、调整、检验都编写成程序，细节见以下找全部解的程序。//Unity3D教程手册：www.unitymanual.com

【程序】

|  |  |  |
| --- | --- | --- |
|  |  |  |

|  |  |
| --- | --- |
| 001 | # include |
| 002 |  |
| 003 | # define N 12 |
| 004 |  |
| 005 | **void** write(**int** a[ ]) |
| 006 |  |
| 007 | { **int** i,j; |
| 008 |  |
| 009 | **for** (i=0;i<3;i++) |
| 010 |  |
| 011 | { **for** (j=0;j<3;j++) |
| 012 |  |
| 013 | printf(“%3d”,a[3\*i+j]); |
| 014 |  |
| 015 | printf(“\n”); |
| 016 |  |
| 017 | } |
| 018 |  |
| 019 | scanf(“%\*c”); |
| 020 |  |
| 021 | } |
| 022 |  |
| 023 | **int** b[N+1]; |
| 024 |  |
| 025 | **int** a[10]; |
| 026 |  |
| 027 | **int** isprime(**int** m) |
| 028 |  |
| 029 | { **int** i; |
| 030 |  |
| 031 | **int** primes[ ]={2,3,5,7,11,17,19,23,29,-1}; |
| 032 |  |
| 033 | **if** (m==1||m%2=0) **return** 0; |
| 034 |  |
| 035 | **for** (i=0;primes[i]>0;i++) |
| 036 |  |
| 037 | **if** (m==primes[i]) **return** 1; |
| 038 |  |
| 039 | **for** (i=3;i\*i<=m;) |
| 040 |  |
| 041 | { **if** (m%i==0) **return** 0; |
| 042 |  |
| 043 | i+=2; |
| 044 |  |
| 045 | } |
| 046 |  |
| 047 | **return** 1; |
| 048 |  |
| 049 | } |
| 050 |  |
| 051 | **int** checkmatrix[ ][3]={ {-1},{0,-1},{1,-1},{0,-1},{1,3,-1}, |
| 052 |  |
| 053 | {2,4,-1},{3,-1},{4,6,-1},{5,7,-1}}; |
| 054 |  |
| 055 | **int** selectnum(**int** start) |
| 056 |  |
| 057 | { **int** j; |
| 058 |  |
| 059 | **for** (j=start;j<=N;j++) |
| 060 |  |
| 061 | **if** (b[j]) **return** j |
| 062 |  |
| 063 | **return** 0; |
| 064 |  |
| 065 | } |
| 066 |  |
| 067 | **int** check(**int** pos) |
| 068 |  |
| 069 | { **int** i,j; |
| 070 |  |
| 071 | **if** (pos<0) **return** 0; |
| 072 |  |
| 073 | **for** (i=0;(j=checkmatrix[pos][i])>=0;i++) |
| 074 |  |
| 075 | **if** (!isprime(a[pos]+a[j]) |
| 076 |  |
| 077 | **return** 0; |
| 078 |  |
| 079 | **return** 1; |
| 080 |  |
| 081 | } |
| 082 |  |
| 083 | **int** extend(**int** pos) |
| 084 |  |
| 085 | { a[++pos]=selectnum(1); |
| 086 |  |
| 087 | b[a][pos]]=0; |
| 088 |  |
| 089 | **return** pos; |
| 090 |  |
| 091 | } |
| 092 |  |
| 093 | **int** change(**int** pos) |
| 094 |  |
| 095 | { **int** j; |
| 096 |  |
| 097 | **while** (pos>=0&&(j=selectnum(a[pos]+1))==0) |
| 098 |  |
| 099 | b[a[pos--]]=1; |
| 100 |  |
| 101 | **if** (pos<0) **return** –1 |
| 102 |  |
| 103 | b[a[pos]]=1; |
| 104 |  |
| 105 | a[pos]=j; |
| 106 |  |
| 107 | b[j]=0; |
| 108 |  |
| 109 | **return** pos; |
| 110 |  |
| 111 | } |
| 112 |  |
| 113 | **void** find() |
| 114 |  |
| 115 | { **int** ok=0,pos=0; |
| 116 |  |
| 117 | a[pos]=1; |
| 118 |  |
| 119 | b[a[pos]]=0; |
| 120 |  |
| 121 | **do** { |
| 122 |  |
| 123 | **if** (ok) |
| 124 |  |
| 125 | **if** (pos==8) |
| 126 |  |
| 127 | { write(a); |
| 128 |  |
| 129 | pos=change(pos); |
| 130 |  |
| 131 | } |
| 132 |  |
| 133 | **else** pos=extend(pos); |
| 134 |  |
| 135 | **else** pos=change(pos); |
| 136 |  |
| 137 | ok=check(pos); |
| 138 |  |
| 139 | } **while** (pos>=0) |
| 140 |  |
| 141 | } |
| 142 |  |
| 143 | **void** main() |
| 144 |  |
| 145 | { **int** i; |
| 146 |  |
| 147 | **for** (i=1;i<=N;i++) |
| 148 |  |
| 149 | b[i]=1; |
| 150 |  |
| 151 | find(); |
| 152 |  |
| 153 | } |
| 154 |  |

【问题】 n皇后问题

问题描述：求出在一个n×n的棋盘上，放置n个不能互相捕捉的国际象棋“皇后”的所有布局。

这是来源于国际象棋的一个问题。皇后可以沿着纵横和两条斜线4个方向相互捕捉。如图所示，一个皇后放在棋盘的第4行第3列位置上，则棋盘上凡打“×”的位置上的皇后就能与这个皇后相互捕捉。

1 2 3 4 5 6 7 8

× ×

× × ×

× × ×

× × Q × × × × ×

× × ×

× × ×

× ×

× ×

从图中可以得到以下启示：一个合适的解应是在每列、每行上只有一个皇后，且一条斜线上也只有一个皇后。

求解过程从空配置开始。在第1列至第m列为合理配置的基础上，再配置第m+1列，直至第n列配置也是合理时，就找到了一个解。接着改变第n列配置，希望获得下一个解。另外，在任一列上，可能有n种配置。开始时配置在第1行，以后改变时，顺次选择第2行、第3行、…、直到第n行。当第n行配置也找不到一个合理的配置时，就要回溯，去改变前一列的配置。得到求解皇后问题的算法如下：

{ 输入棋盘大小值n；

|  |  |  |
| --- | --- | --- |
|  |  |  |

|  |  |
| --- | --- |
| 01 | m=0; |
| 02 |  |
| 03 | good=1; |
| 04 |  |
| 05 | **do** { |
| 06 |  |
| 07 | **if** (good) |
| 08 |  |
| 09 | **if** (m==n) |
| 10 |  |
| 11 | { 输出解； |
| 12 |  |
| 13 | 改变之，形成下一个候选解; |
| 14 |  |
| 15 | } |
| 16 |  |
| 17 | **else** 扩展当前候选接至下一列； |
| 18 |  |
| 19 | **else** 改变之，形成下一个候选解； |
| 20 |  |
| 21 | good=检查当前候选解的合理性； |
| 22 |  |
| 23 | } **while** (m!=0); |
| 24 |  |
| 25 | } |
| 26 |  |

在编写程序之前，先确定边式棋盘的数据结构。比较直观的方法是采用一个二维数组，但仔细观察就会发现，这种表示方法给调整候选解及检查其合理性带来困难。更好的方法乃是尽可能直接表示那些常用的信息。对于本题来说，“常用信息”并不是皇后的具体位置，而是“一个皇后是否已经在某行和某条斜线合理地安置好了”。因在某一列上恰好放一个皇后，引入一个一维数组（col[ ]），值col[i]表示在棋盘第i列、col[i]行有一个皇后。例如：col[3]=4，就表示在棋盘的第3列、第4行上有一个皇后。另外，为了使程序在找完了全部解后回溯到最初位置，设定col[0]的初值为0当回溯到第0列时，说明程序已求得全部解，结束程序运行。//Unity3D教程手册：www.unitymanual.com

为使程序在检查皇后配置的合理性方面简易方便，引入以下三个工作数组：

（1） 数组a[ ]，a[k]表示第k行上还没有皇后；

（2） 数组b[ ]，b[k]表示第k列右高左低斜线上没有皇后；

（3） 数组 c[ ]，c[k]表示第k列左高右低斜线上没有皇后；

棋盘中同一右高左低斜线上的方格，他们的行号与列号之和相同；同一左高右低斜线上的方格，他们的行号与列号之差均相同。

初始时，所有行和斜线上均没有皇后，从第1列的第1行配置第一个皇后开始，在第m列col[m]行放置了一个合理的皇后后，准备考察第m+1列时，在数组a[ ]、b[ ]和c[ ]中为第m列，col[m]行的位置设定有皇后标志；当从第m列回溯到第m-1列，并准备调整第m-1列的皇后配置时，清除在数组a[ ]、b[ ]和c[ ]中设置的关于第m-1列，col[m-1]行有皇后的标志。一个皇后在m列，col[m]行方格内配置是合理的，由数组a[ ]、b[ ]和c[ ]对应位置的值都为1来确定。细节见以下程序：

【程序】

|  |  |  |
| --- | --- | --- |
|  |  |  |

|  |  |
| --- | --- |
| 01 | # include |
| 02 |  |
| 03 | # include |
| 04 |  |
| 05 | # define MAXN 20 |
| 06 |  |
| 07 | **int** n,m,good; |
| 08 |  |
| 09 | **int** col[MAXN+1],a[MAXN+1],b[2\*MAXN+1],c[2\*MAXN+1]; |
| 10 |  |
| 11 | **void** main() |
| 12 |  |
| 13 | { **int** j; |
| 14 |  |
| 15 | **char** awn; |
| 16 |  |
| 17 | printf(“Enter n: ”); scanf(“%d”,&n); |
| 18 |  |
| 19 | **for** (j=0;j<=n;j++) a[j]=1; |
| 20 |  |
| 21 | **for** (j=0;j<=2\*n;j++) cb[j]=c[j]=1; |
| 22 |  |
| 23 | m=1; col[1]=1; good=1; col[0]=0; |
| 24 |  |
| 25 | **do** { |
| 26 |  |
| 27 | **if** (good) |
| 28 |  |
| 29 | **if** (m==n) |
| 30 |  |
| 31 | { printf(“列\t行”); |
| 32 |  |
| 33 | **for** (j=1;j<=n;j++) |
| 34 |  |
| 35 | printf(“%3d\t%d\n”,j,col[j]); |
| 36 |  |
| 37 | printf(“Enter a character (Q/q **for** exit)!\n”); |
| 38 |  |
| 39 | scanf(“%c”,&awn); |
| 40 |  |
| 41 | **if** (awn=='Q'||awn=='q') exit(0); |
| 42 |  |
| 43 | **while** (col[m]==n) |
| 44 |  |
| 45 | { m--; |
| 46 |  |
| 47 | a[col[m]]=b[m+col[m]]=c[n+m-col[m]]=1; |
| 48 |  |
| 49 | } |
| 50 |  |
| 51 | col[m]++; |
| 52 |  |
| 53 | } |
| 54 |  |
| 55 | **else** |
| 56 |  |
| 57 | { a[col[m]]=b[m+col[m]]=c[n+m-col[m]]=0; |
| 58 |  |
| 59 | col[++m]=1; |
| 60 |  |
| 61 | } |
| 62 |  |
| 63 | **else** |
| 64 |  |
| 65 | { **while** (col[m]==n) |
| 66 |  |
| 67 | { m--; |
| 68 |  |
| 69 | a[col[m]]=b[m+col[m]]=c[n+m-col[m]]=1; |
| 70 |  |
| 71 | } |
| 72 |  |
| 73 | col[m]++; |
| 74 |  |
| 75 | } |
| 76 |  |
| 77 | good=a[col[m]]&&b[m+col[m]]&&c[n+m-col[m]]; |
| 78 |  |
| 79 | } **while** (m!=0); |
| 80 |  |
| 81 | } |

试探法找解算法也常常被编写成递归函数，下面两程序中的函数queen\_all()和函数queen\_one()能分别用来解皇后问题的全部解和一个解。

【程序】

|  |  |  |
| --- | --- | --- |
|  |  |  |

|  |  |
| --- | --- |
| 01 | # include |
| 02 |  |
| 03 | # include |
| 04 |  |
| 05 | # define MAXN 20 |
| 06 |  |
| 07 | **int** n; |
| 08 |  |
| 09 | **int** col[MAXN+1],a[MAXN+1],b[2\*MAXN+1],c[2\*MAXN+1]; |
| 10 |  |
| 11 | **void** main() |
| 12 |  |
| 13 | { **int** j; |
| 14 |  |
| 15 | printf(“Enter n: ”); scanf(“%d”,&n); |
| 16 |  |
| 17 | **for** (j=0;j<=n;j++) a[j]=1; |
| 18 |  |
| 19 | **for** (j=0;j<=2\*n;j++) cb[j]=c[j]=1; |
| 20 |  |
| 21 | queen\_all(1,n); |
| 22 |  |
| 23 | } |
| 24 |  |
| 25 | **void** queen\_all(**int** k,**int** n) |
| 26 |  |
| 27 | { **int** i,j; |
| 28 |  |
| 29 | **char** awn; |
| 30 |  |
| 31 | **for** (i=1;i<=n;i++) |
| 32 |  |
| 33 | **if** (a[i]&&b[k+i]&&c[n+k-i]) |
| 34 |  |
| 35 | { col[k]=i; |
| 36 |  |
| 37 | a[i]=b[k+i]=c[n+k-i]=0; |
| 38 |  |
| 39 | **if** (k==n) |
| 40 |  |
| 41 | { printf(“列\t行”); |
| 42 |  |
| 43 | **for** (j=1;j<=n;j++) |
| 44 |  |
| 45 | printf(“%3d\t%d\n”,j,col[j]); |
| 46 |  |
| 47 | printf(“Enter a character (Q/q **for** exit)!\n”); |
| 48 |  |
| 49 | scanf(“%c”,&awn); |
| 50 |  |
| 51 | **if** (awn=='Q'||awn=='q') exit(0); |
| 52 |  |
| 53 | } |
| 54 |  |
| 55 | queen\_all(k+1,n); |
| 56 |  |
| 57 | a[i]=b[k+i]=c[n+k-i]; |
| 58 |  |
| 59 | } |
| 60 |  |
| 61 | } |
| 62 |  |

采用递归方法找一个解与找全部解稍有不同，在找一个解的算法中，递归算法要对当前候选解最终是否能成为解要有回答。当它成为最终解时，递归函数就不再递归试探，立即返回；若不能成为解，就得继续试探。设函数queen\_one()返回1表示找到解，返回0表示当前候选解不能成为解。细节见以下函数。

【程序】

|  |  |  |
| --- | --- | --- |
|  |  |  |

|  |  |
| --- | --- |
| 01 | # define MAXN 20 |
| 02 |  |
| 03 | **int** n; |
| 04 |  |
| 05 | **int** col[MAXN+1],a[MAXN+1],b[2\*MAXN+1],c[2\*MAXN+1]; |
| 06 |  |
| 07 | **int** queen\_one(**int** k,**int** n) |
| 08 |  |
| 09 | { **int** i,found; |
| 10 |  |
| 11 | i=found=0; |
| 12 |  |
| 13 | **While** (!found&&i　　 { i++; |
| 14 |  |
| 15 | **if** (a[i]&&b[k+i]&&c[n+k-i]) |
| 16 |  |
| 17 | { col[k]=i; |
| 18 |  |
| 19 | a[i]=b[k+i]=c[n+k-i]=0; |
| 20 |  |
| 21 | **if** (k==n) **return** 1; |
| 22 |  |
| 23 | **else** |
| 24 |  |
| 25 | found=queen\_one(k+1,n); |
| 26 |  |
| 27 | a[i]=b[k+i]=c[n+k-i]=1; |
| 28 |  |
| 29 | } |
| 30 |  |
| 31 | } |
| 32 |  |
| 33 | **return** found; |
| 34 |  |
| 35 | } |