**Unity3D教程：游戏开发算法（三）**

Posted on 2013年06月26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 60 次

六、贪婪法

贪婪法是一种不追求最优解，只希望得到较为满意解的方法。贪婪法一般可以快速得到满意的解，因为它省去了为找最优解要穷尽所有可能而必须耗费的大量时间。贪婪法常以当前情况为基础作最优选择，而不考虑各种可能的整体情况，所以贪婪法不要回溯。

例如平时购物找钱时，为使找回的零钱的硬币数最少，不考虑找零钱的所有各种发表方案，而是从最大面值的币种开始，按递减的顺序考虑各币种，先尽量用大面值 的币种，当不足大面值币种的金额时才去考虑下一种较小面值的币种。这就是在使用贪婪法。这种方法在这里总是最优，是因为银行对其发行的硬币种类和硬币面值 的巧妙安排。如只有面值分别为1、5和11单位的硬币，而希望找回总额为15单位的硬币。按贪婪算法，应找1个11单位面值的硬币和4个1单位面值的硬 币，共找回5个硬币。但最优的解应是3个5单位面值的硬币。

【问题】 装箱问题

问题描述：装箱问题可简述如下：设有编号为0、1、…、n-1的n种物品，体积分别为v0、v1、…、vn-1。将这n种物品装到容量都为V的若干箱 子里。约定这n种物品的体积均不超过V，即对于0≤i＜n，有0＜vi≤V。不同的装箱方案所需要的箱子数目可能不同。装箱问题要求使装尽这n种物品的箱 子数要少。

若考察将n种物品的集合分划成n个或小于n个物品的所有子集，最优解就可以找到。但所有可能划分的总数太大。对适当大的n，找出所有可能的划分要花费的时 间是无法承受的。为此，对装箱问题采用非常简单的近似算法，即贪婪法。该算法依次将物品放到它第一个能放进去的箱子中，该算法虽不能保证找到最优解，但还 是能找到非常好的解。不失一般性，设n件物品的体积是按从大到小排好序的，即有v0≥v1≥…≥vn-1。如不满足上述要求，只要先对这n件物品按它们的 体积从大到小排序，然后按排序结果对物品重新编号即可。装箱算法简单描述如下：

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

|  |  |
| --- | --- |
| 01 | { 输入箱子的容积； |
| 02 |  |
| 03 | 输入物品种数n； |
| 04 |  |
| 05 | 按体积从大到小顺序，输入各物品的体积； |
| 06 |  |
| 07 | 预置已用箱子链为空； |
| 08 |  |
| 09 | 预置已用箱子计数器box\_count为0； |
| 10 |  |
| 11 | **for** (i=0;i　　 { 从已用的第一只箱子开始顺序寻找能放入物品i 的箱子j； |
| 12 |  |
| 13 | **if** （已用箱子都不能再放物品i） |
| 14 |  |
| 15 | { 另用一个箱子，并将物品i放入该箱子； |
| 16 |  |
| 17 | box\_count++； |
| 18 |  |
| 19 | } |
| 20 |  |
| 21 | **else** |
| 22 |  |
| 23 | 将物品i放入箱子j； |
| 24 |  |
| 25 | } |
| 26 |  |
| 27 | } |
| 28 |  |

上述算法能求出需要的箱子数box\_count，并能求出各箱子所装物品。下面的例子说明该算法不一定能找到最优解，设有6种物品，它们的体积分别为： 60、45、35、20、20和20单位体积，箱子的容积为100个单位体积。按上述算法计算，需三只箱子，各箱子所装物品分别为：第一只箱子装物品1、 3；第二只箱子装物品2、4、5；第三只箱子装物品6。而最优解为两只箱子，分别装物品1、4、5和2、3、6。

若每只箱子所装物品用链表来表示，链表首结点指针存于一个结构中，结构记录尚剩余的空间量和该箱子所装物品链表的首指针。另将全部箱子的信息也构成链表。以下是按以上算法编写的程序。

【程序】

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

|  |  |
| --- | --- |
| 001 | # include |
| 002 |  |
| 003 | # include |
| 004 |  |
| 005 | typedef **struct** ele |
| 006 |  |
| 007 | { **int** vno; |
| 008 |  |
| 009 | **struct** ele \*link; |
| 010 |  |
| 011 | } ELE; |
| 012 |  |
| 013 | typedef **struct** hnode |
| 014 |  |
| 015 | { **int** remainder; |
| 016 |  |
| 017 | ELE \*head; |
| 018 |  |
| 019 | **Struct** hnode \*next; |
| 020 |  |
| 021 | } HNODE; |
| 022 |  |
| 023 | **void** main() |
| 024 |  |
| 025 | { **int** n, i, box\_count, box\_volume, \*a; |
| 026 |  |
| 027 | HNODE \*box\_h, \*box\_t, \*j; |
| 028 |  |
| 029 | ELE \*p, \*q; |
| 030 |  |
| 031 | Printf(“输入箱子容积\n”); |
| 032 |  |
| 033 | Scanf(“%d”,&box\_volume); |
| 034 |  |
| 035 | Printf(“输入物品种数\n”); |
| 036 |  |
| 037 | Scanf(“%d”,&n); |
| 038 |  |
| 039 | A=(**int** \*)malloc(sizeof(**int**)\*n); |
| 040 |  |
| 041 | Printf(“请按体积从大到小顺序输入各物品的体积：”); |
| 042 |  |
| 043 | **For** (i=0;i　　 Box\_h=box\_t=**NULL**; |
| 044 |  |
| 045 | Box\_count=0; |
| 046 |  |
| 047 | **For** (i=0;i　　 { p=(ELE \*)malloc(sizeof(ELE)); |
| 048 |  |
| 049 | p->vno=i; |
| 050 |  |
| 051 | **for** (j=box\_h;j!=**NULL**;j=j->next) |
| 052 |  |
| 053 | **if** (j->remainder>=a[i]) **break**; |
| 054 |  |
| 055 | **if** (j==**NULL**) |
| 056 |  |
| 057 | { j=(HNODE \*)malloc(sizeof(HNODE)); |
| 058 |  |
| 059 | j->remainder=box\_volume-a[i]; |
| 060 |  |
| 061 | j->head=**NULL**; |
| 062 |  |
| 063 | **if** (box\_h==**NULL**) box\_h=box\_t=j; |
| 064 |  |
| 065 | **else** box\_t=boix\_t->next=j; |
| 066 |  |
| 067 | j->next=**NULL**; |
| 068 |  |
| 069 | box\_count++; |
| 070 |  |
| 071 | } |
| 072 |  |
| 073 | **else** j->remainder-=a[i]; |
| 074 |  |
| 075 | **for** (q=j->next;q!=**NULL**&&q->link!=**NULL**;q=q->link); |
| 076 |  |
| 077 | **if** (q==**NULL**) |
| 078 |  |
| 079 | { p->link=j->head; |
| 080 |  |
| 081 | j->head=p; |
| 082 |  |
| 083 | } |
| 084 |  |
| 085 | **else** |
| 086 |  |
| 087 | { p->link=**NULL**; |
| 088 |  |
| 089 | q->link=p; |
| 090 |  |
| 091 | } |
| 092 |  |
| 093 | } |
| 094 |  |
| 095 | printf(“共使用了%d只箱子”，box\_count); |
| 096 |  |
| 097 | printf(“各箱子装物品情况如下：”); |
| 098 |  |
| 099 | **for** (j=box\_h,i=1;j!=**NULL**;j=j->next,i++) |
| 100 |  |
| 101 | { printf(“第%2d只箱子，还剩余容积%4d，所装物品有；\n”,I,j->remainder); |
| 102 |  |
| 103 | **for** (p=j->head;p!=**NULL**;p=p->link) |
| 104 |  |
| 105 | printf(“%4d”,p->vno+1); |
| 106 |  |
| 107 | printf(“\n”); |
| 108 |  |
| 109 | } |
| 110 |  |
| 111 | } |
| 112 |  |

【问题】 马的遍历

问题描述：在8×8方格的棋盘上，从任意指定的方格出发，为马寻找一条走遍棋盘每一格并且只经过一次的一条路径。

马在某个方格，可以在一步内到达的不同位置最多有8个，如图所示。如用二维数组board[ ][ ]表示棋盘，其元素记录马经过该位置时的步骤号。另对马的8种可能走法（称为着法）设定一个顺序，如当前位置在棋盘的（i，j）方格，下一个可能的位置依 次为（i+2，j+1）、（i+1，j+2）、（i-1，j+2）、（i-2，j+1）、（i-2，j-1）、（i-1，j-2）、（i+1，j-2）、 （i+2，j-1），实际可以走的位置尽限于还未走过的和不越出边界的那些位置。为便于程序的同意处理，可以引入两个数组，分别存储各种可能走法对当前位 置的纵横增量。

4 3

5 2

马

6 1

7 0

对于本题，一般可以采用回溯法，这里采用Warnsdoff策略求解，这也是一种贪婪法，其选择下一出口的贪婪标准是在那些允许走的位置中，选择出口最少 的那个位置。如马的当前位置（i，j）只有三个出口，他们是位置（i+2，j+1）、（i-2，j+1）和（i-1，j-2），如分别走到这些位置，这三 个位置又分别会有不同的出口，假定这三个位置的出口个数分别为4、2、3，则程序就选择让马走向（i-2，j+1）位置。

由于程序采用的是一种贪婪法，整个找解过程是一直向前，没有回溯，所以能非常快地找到解。但是，对于某些开始位置，实际上有解，而该算法不能找到解。对于 找不到解的情况，程序只要改变8种可能出口的选择顺序，就能找到解。改变出口选择顺序，就是改变有相同出口时的选择标准。以下程序考虑到这种情况，引入变 量start，用于控制8种可能着法的选择顺序。开始时为0，当不能找到解时，就让start增1，重新找解。细节以下程序。

【程序】

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

|  |  |
| --- | --- |
| 001 | # include |
| 002 |  |
| 003 | **int** delta\_i[ ]={2,1,-1,-2,-2,-1,1,2}; |
| 004 |  |
| 005 | **int** delta\_j[ ]={1,2,2,1,-1,-2,-2,-1}; |
| 006 |  |
| 007 | **int** board[8][8]; |
| 008 |  |
| 009 | **int** exitn(**int** i,**int** j,**int** s,**int** a[ ]) |
| 010 |  |
| 011 | { **int** i1,j1,k,count; |
| 012 |  |
| 013 | **for** (count=k=0;k<8;k++) |
| 014 |  |
| 015 | { i1=i+delta\_i[(s+k)%8]; |
| 016 |  |
| 017 | j1=i+delta\_j[(s+k)%8]; |
| 018 |  |
| 019 | **if** (i1>=0&&i1<8&&j1>=0&&j1<8&&board[I1][j1]==0) |
| 020 |  |
| 021 | a[count++]=(s+k)%8; |
| 022 |  |
| 023 | } |
| 024 |  |
| 025 | **return** count; |
| 026 |  |
| 027 | } |
| 028 |  |
| 029 | **int** next(**int** i,**int** j,**int** s) |
| 030 |  |
| 031 | { **int** m,k,mm,min,a[8],b[8],temp; |
| 032 |  |
| 033 | m=exitn(i,j,s,a); |
| 034 |  |
| 035 | **if** (m==0) **return** –1; |
| 036 |  |
| 037 | **for** (min=9,k=0;k　　 { temp=exitn(I+delta\_i[a[k]],j+delta\_j[a[k]],s,b); |
| 038 |  |
| 039 | **if** (temp　　 { min=temp; |
| 040 |  |
| 041 | kk=a[k]; |
| 042 |  |
| 043 | } |
| 044 |  |
| 045 | } |
| 046 |  |
| 047 | **return** kk; |
| 048 |  |
| 049 | } |
| 050 |  |
| 051 | **void** main() |
| 052 |  |
| 053 | { **int** sx,sy,i,j,step,no,start; |
| 054 |  |
| 055 | **for** (sx=0;sx<8;sx++) |
| 056 |  |
| 057 | **for** (sy=0;sy<8;sy++) |
| 058 |  |
| 059 | { start=0; |
| 060 |  |
| 061 | **do** { |
| 062 |  |
| 063 | **for** (i=0;i<8;i++) |
| 064 |  |
| 065 | **for** (j=0;j<8;j++) |
| 066 |  |
| 067 | board[i][j]=0; |
| 068 |  |
| 069 | board[sx][sy]=1; |
| 070 |  |
| 071 | I=sx; j=sy; |
| 072 |  |
| 073 | **For** (step=2;step<64;step++) |
| 074 |  |
| 075 | { **if** ((no=next(i,j,start))==-1) **break**; |
| 076 |  |
| 077 | I+=delta\_i[no]; |
| 078 |  |
| 079 | j+=delta\_j[no]; |
| 080 |  |
| 081 | board[i][j]=step; |
| 082 |  |
| 083 | } |
| 084 |  |
| 085 | **if** (step>64) **break**; |
| 086 |  |
| 087 | start++; |
| 088 |  |
| 089 | } **while**(step<=64) |
| 090 |  |
| 091 | **for** (i=0;i<8;i++) |
| 092 |  |
| 093 | { **for** (j=0;j<8;j++) |
| 094 |  |
| 095 | printf(“%4d”,board[i][j]); |
| 096 |  |
| 097 | printf(“\n\n”); |
| 098 |  |
| 099 | } |
| 100 |  |
| 101 | scanf(“%\*c”); |
| 102 |  |
| 103 | } |
| 104 |  |
| 105 | } |
| 106 |  |

七、分治法

1、分治法的基本思想

任何一个可以用计算机求解的问题所需的计算时间都与其规模N有关。问题的规模越小，越容易直接求解，解题所需的计算时间也越少。例如，对于n个元素的 排序问题，当n=1时，不需任何计算；n=2时，只要作一次比较即可排好序；n=3时只要作3次比较即可，…。而当n较大时，问题就不那么容易处理了。要 想直接解决一个规模较大的问题，有时是相当困难的。

分治法的设计思想是，将一个难以直接解决的大问题，分割成一些规模较小的相同问题，以便各个击破，分而治之。

如果原问题可分割成k个子问题

2、分治法的适用条件

分治法所能解决的问题一般具有以下几个特征：

（1）该问题的规模缩小到一定的程度就可以容易地解决；

（2）该问题可以分解为若干个规模较小的相同问题，即该问题具有最优子结构性质；

（3）利用该问题分解出的子问题的解可以合并为该问题的解；

（4）该问题所分解出的各个子问题是相互独立的，即子问题之间不包含公共的子子问题。

上述的第一条特征是绝大多数问题都可以满足的，因为问题的计算复杂性一般是随着问题规模的增加而增加；第二条特征是应用分治法的前提，它也是大多数问 题可以满足的，此特征反映了递归思想的应用；第三条特征是关键，能否利用分治法完全取决于问题是否具有第三条特征，如果具备了第一条和第二条特征，而不具 备第三条特征，则可以考虑贪心法或动态规划法。第四条特征涉及到分治法的效率，如果各子问题是不独立的，则分治法要做许多不必要的工作，重复地解公共的子 问题，此时虽然可用分治法，但一般用动态规划法较好。

3、分治法的基本步骤

分治法在每一层递归上都有三个步骤：

（1）分解：将原问题分解为若干个规模较小，相互独立，与原问题形式相同的子问题；

（2）解决：若子问题规模较小而容易被解决则直接解，否则递归地解各个子问题；

（3）合并：将各个子问题的解合并为原问题的解。

它的一般的算法设计模式如下：

Divide\_and\_Conquer（P）

if |P|≤n0

then return（ADHOC（P））

将P分解为较小的子问题P1、P2、…、Pk

for i←1 to k

do

yi ← Divide-and-Conquer（Pi） △ 递归解决Pi

T ← MERGE（y1，y2，…，yk） △ 合并子问题

Return（T）

其中 |P| 表示问题P的规模；n0为一阈值，表示当问题P的规模不超过n0时，问题已容易直接解出，不必再继续分解。ADHOC（P）是该分治法中的基本子算法，用于直接解小规模的问题P。因此，当P的规模不超过n0时，直接用算法ADHOC（P）求解。

算法MERGE（y1，y2，…，yk）是该分治法中的合并子算法，用于将P的子问题P1、P2、…、Pk的相应的解y1、y2、…、yk合并为P的解。

根据分治法的分割原则，原问题应该分为多少个子问题才较适宜？各个子问题的规模应该怎样才为适当？这些问题很难予以肯定的回答。但人们从大量实践中发 现，在用分治法设计算法时，最好使子问题的规模大致相同。换句话说，将一个问题分成大小相等的k个子问题的处理方法是行之有效的。许多问题可以取k=2。 这种使子问题规模大致相等的做法是出自一种平衡子问题的思想，它几乎总是比子问题规模不等的做法要好。

分治法的合并步骤是算法的关键所在。有些问题的合并方法比较明显，有些问题合并方法比较复杂，或者是有多种合并方案；或者是合并方案不明显。究竟应该怎样合并，没有统一的模式，需要具体问题具体分析。

【问题】 大整数乘法

问题描述：

通常，在分析一个算法的计算复杂性时，都将加法和乘法运算当作是基本运算来处理，即将执行一次加法或乘法运算所需的计算时间当作一个仅取决于计算机硬件处理速度的常数。

这个假定仅在计算机硬件能对参加运算的整数直接表示和处理时才是合理的。然而，在某些情况下，我们要处理很大的整数，它无法在计算机硬件能直接表示的 范围内进行处理。若用浮点数来表示它，则只能近似地表示它的大小，计算结果中的有效数字也受到限制。若要精确地表示大整数并在计算结果中要求精确地得到所 有位数上的数字，就必须用软件的方法来实现大整数的算术运算。

请设计一个有效的算法，可以进行两个n位大整数的乘法运算。

设X和Y都是n位的二进制整数，现在要计算它们的乘积XY。我们可以用小学所学的方法来设计一个计算乘积XY的算法，但是这样做计算步骤太多，显得效 率较低。如果将每2个1位数的乘法或加法看作一步运算，那么这种方法要作O(n2)步运算才能求出乘积XY。下面我们用分治法来设计一个更有效的大整数乘 积算法。

图6-3 大整数X和Y的分段

我们将n位的二进制整数X和Y各分为2段，每段的长为n/2位（为简单起见，假设n是2的幂），如图6-3所示。

由此，X=A2n/2+B，Y=C2n/2+D。这样，X和Y的乘积为：

XY=(A2n/2+B)(C2n/2+D)=AC2n+(AD+CB)2n/2+BD （1）

如果按式（1）计算XY，则我们必须进行4次n/2位整数的乘法(AC，AD，BC和BD)，以及3次不超过n位的整数加法（分别对应于式（1）中的 加号），此外还要做2次移位（分别对应于式（1）中乘2n和乘2n/2）。所有这些加法和移位共用O（n）步运算。设T（n）是2个n位整数相乘所需的运 算总数，则由式（1），我们有：

（2）由此可得T（n）=O（n2）。因此，用（1）式来计算X和Y的乘积并不比小学生的方法更有效。要想改进算法的计算复杂性，必须减少乘法次数。为此我们把XY写成另一种形式：

XY=AC2n+[(A-B)(D-C)+AC+BD]2n/2+BD （3）

虽然，式（3）看起来比式（1）复杂些，但它仅需做3次n/2位整数的乘法（AC，BD和（A-B）（D-C）），6次加、减法和2次移位。由此可得：

（4）

用解递归方程的套用公式法马上可得其解为T(n)=O(nlog3)=O(n1.59)。利用式（3），并考虑到X和Y的符号对结果的影响，我们给出大整数相乘的完整算法MULT如下：

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

|  |  |
| --- | --- |
| 01 | function MULT(X，Y，n); {X和Y为2个小于2n的整数，返回结果为X和Y的乘积XY} |
| 02 |  |
| 03 | begin |
| 04 |  |
| 05 | S=SIGN(X)\*SIGN(Y); {S为X和Y的符号乘积} |
| 06 |  |
| 07 | X=ABS(X); |
| 08 |  |
| 09 | Y=ABS(Y); {X和Y分别取绝对值} |
| 10 |  |
| 11 | **if** n=1 then |
| 12 |  |
| 13 | **if** (X=1)and(Y=1) then **return**(S) |
| 14 |  |
| 15 | **else** **return**(0) |
| 16 |  |
| 17 | **else** begin |
| 18 |  |
| 19 | A=X的左边n/2位; |
| 20 |  |
| 21 | B=X的右边n/2位; |
| 22 |  |
| 23 | C=Y的左边n/2位; |
| 24 |  |
| 25 | D=Y的右边n/2位; |
| 26 |  |
| 27 | ml=MULT(A,C,n/2); |
| 28 |  |
| 29 | m2=MULT(A-B,D-C,n/2); |
| 30 |  |
| 31 | m3=MULT(B,D,n/2); |
| 32 |  |
| 33 | S=S\*(m1\*2n+(m1+m2+m3)\*2n/2+m3); |
| 34 |  |
| 35 | **return**(S); |
| 36 |  |
| 37 | end; |
| 38 |  |
| 39 | end; |

上述二进制大整数乘法同样可应用于十进制大整数的乘法以提高乘法的效率减少乘法次数。

【问题】 最接近点对问题

问题描述：

在应用中，常用诸如点、圆等简单的几何对象代表现实世界中的实体。在涉及这些几何对象的问题中，常需要了解其邻域中其他几何对象的信息。例如，在空中 交通控制问题中，若将飞机作为空间中移动的一个点来看待，则具有最大碰撞危险的2架飞机，就是这个空间中最接近的一对点。这类问题是计算几何学中研究的基 本问题之一。下面我们着重考虑平面上的最接近点对问题。

最接近点对问题的提法是：给定平面上n个点，找其中的一对点，使得在n个点的所有点对中，该点对的距离最小。

严格地说，最接近点对可能多于1对。为了简单起见，这里只限于找其中的一对。

这个问题很容易理解，似乎也不难解决。我们只要将每一点与其他n-1个点的距离算出，找出达到最小距离的两个点即可。然而，这样做效率太低，需要O(n2)的计算时间。我们能否找到问题的一个O (nlogn)算法。

这个问题显然满足分治法的第一个和第二个适用条件，我们考虑将所给的平面上n个点的集合S分成2个子集S1和S2，每个子集中约有n/2个点，然后在 每个子集中递归地求其最接近的点对。在这里，一个关键的问题是如何实现分治法中的合并步骤，即由S1和S2的最接近点对，如何求得原集合S中的最接近点 对，因为S1和S2的最接近点对未必就是S的最接近点对。如果组成S的最接近点对的2个点都在S1中或都在S2中，则问题很容易解决。但是，如果这2个点 分别在S1和S2中，则对于S1中任一点p，S2中最多只有n/2个点与它构成最接近点对的候选者，仍需做n2/4次计算和比较才能确定S的最接近点对。 因此，依此思路，合并步骤耗时为O(n2)。整个算法所需计算时间T(n)应满足：

T(n)=2T(n/2)+O(n2)

它的解为T(n)=O(n2)，即与合并步骤的耗时同阶，显示不出比用穷举的方法好。从解递归方程的套用公式法，我们看到问题出在合并步骤耗时太多。这启发我们把注意力放在合并步骤上。

为了使问题易于理解和分析，我们先来考虑一维的情形。此时S中的n个点退化为x轴上的n个实数x1、x2、…、xn。最接近点对即为这n个实数中相差 最小的2个实数。我们显然可以先将x1、x2、…、xn排好序，然后，用一次线性扫描就可以找出最接近点对。这种方法主要计算时间花在排序上，因此如在排 序算法中所证明的，耗时为O(nlogn)。然而这种方法无法直接推广到二维的情形。因此，对这种一维的简单情形，我们还是尝试用分治法来求解，并希望能 推广到二维的情形。

假设我们用x轴上某个点m将S划分为2个子集S1和S2，使得S1={x∈S | x≤m}；S2={x∈S | x>m}。这样一来，对于所有p∈S1和q∈S2有p　　递归地在S1和S2上找出其最接近点对{p1，p2}和{q1，q2}，并设δ=min {|p1-p2|，|q1-q2|}，S中的最接近点对或者是{p1，p2}，或者是{q1，q2}，或者是某个{p3，q3}，其中p3∈S1且 q3∈S2。如图1所示。

图1 一维情形的分治法

我们注意到，如果S的最接近点对是{p3，q3}，即 | p3-q3 | < δ，则p3和q3两者与m的距离不超过δ，即 | p3-m | < δ，| q3-m | < δ，也就是说，p3∈(m-δ，m)，q3∈(m，m+δ)。由于在S1中，每个长度为δ的半闭区间至多包含一个点（否则必有两点距离小于δ），并且m是 S1和S2的分割点，因此(m-δ，m)中至多包含S中的一个点。同理，(m，m+δ)中也至多包含S中的一个点。由图1可以看出，如果(m-δ，m)中 有S中的点，则此点就是S1中最大点。同理，如果(m，m+δ)中有S中的点，则此点就是S2中最小点。因此，我们用线性时间就能找到区间(m-δ，m) 和(m，m+δ)中所有点，即p3和q3。从而我们用线性时间就可以将S1的解和S2的解合并成为S的解。也就是说，按这种分治策略，合并步可在O(n) 时间内完成。这样是否就可以得到一个有效的算法了呢？ 还有一个问题需要认真考虑，即分割点m的选取，及S1和S2的划分。选取分割点m的一个基本要求是由此导出集合S的一个线性分割，即S=S1∪S2 ，S1∩S2=Φ，且S1 {x | x≤m}；S2 {x | x>m}。容易看出，如果选取m=[max（S）+min（S）]/2，可以满足线性分割的要求。选取分割点后，再用O(n)时间即可将S划分成 S1={x∈S | x≤m}和S2={x∈S | x>m}。然而，这样选取分割点m，有可能造成划分出的子集S1和S2的不平衡。例如在最坏情况下，|S1|=1，|S2|=n-1，由此产生的分 治法在最坏情况下所需的计算时间T（n）应满足递归方程：

T（n）=T（n-1）+O（n）

它的解是T（n）=O（n2）。这种效率降低的现象可以通过分治法中“平衡子问题”的方法加以解决。也就是说，我们可以通过适当选择分割点m，使S1 和S2中有大致相等个数的点。自然地，我们会想到用S的n个点的坐标的中位数来作分割点。在选择算法中介绍的选取中位数的线性时间算法使我们可以在O （n）时间内确定一个平衡的分割点m。

至此，我们可以设计出一个求一维点集S中最接近点对的距离的算法pair如下。

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

|  |  |
| --- | --- |
| 01 | **Float** pair（S）; |
| 02 |  |
| 03 | { **if** | S | =2 δ= | x[2]－x[1] | */\*x[1..n]存放的是S中n个点的坐标\*/* |
| 04 |  |
| 05 | **else** |
| 06 |  |
| 07 | { **if** ( | S | =1) δ=∞ |
| 08 |  |
| 09 | **else** |
| 10 |  |
| 11 | { m=S中各点的坐标值的中位数; |
| 12 |  |
| 13 | 构造S1和S2，使S1={x∈S | x&le;m}，S2={x∈S | x＞m}; |
| 14 |  |
| 15 | δ1=pair(S1); |
| 16 |  |
| 17 | δ2=pair(S2); |
| 18 |  |
| 19 | p=max(S1); |
| 20 |  |
| 21 | q=min(S2); |
| 22 |  |
| 23 | δ=min(δ1，δ2，q-p); |
| 24 |  |
| 25 | } |
| 26 |  |
| 27 | **return**(δ); |
| 28 |  |
| 29 | } |

由以上的分析可知，该算法的分割步骤和合并步骤总共耗时O(n)。因此，算法耗费的计算时间T(n)满足递归方程：

解此递归方程可得T(n)=O(nlogn)。

【问题】循环赛日程表

问题描述：设有n=2k个运动员要进行网球循环赛。现要设计一个满足以下要求的比赛日程表：

（1）每个选手必须与其他n-1个选手各赛一次；

（2）每个选手一天只能参赛一次；

（3）循环赛在n-1天内结束。

请按此要求将比赛日程表设计成有n行和n-1列的一个表。在表中的第i行，第j列处填入第i个选手在第j天所遇到的选手。其中1≤i≤n，1≤j≤n-1。

按分治策略，我们可以将所有的选手分为两半，则n个选手的比赛日程表可以通过n/2个选手的比赛日程表来决定。递归地用这种一分为二的策略对选手进行划分，直到只剩下两个选手时，比赛日程表的制定就变得很简单。这时只要让这两个选手进行比赛就可以了。

1 2 3 4 5 6 7

1 2 3 4 5 6 7 8

2 1 4 3 6 7 8 5

3 4 1 2 7 8 5 6

1 2 3 4 3 2 1 8 5 6 7

1 2 3 4 5 6 7 8 1 4 3 2

1 2 1 4 3 6 5 8 7 2 1 4 3

1 2 3 4 1 2 7 8 5 6 3 2 1 4

2 1 4 3 2 1 8 7 6 5 4 3 2 1

（1） （2） （3）

图1 2个、4个和8个选手的比赛日程表

图1所列出的正方形表（3）是8个选手的比赛日程表。其中左上角与左下角的两小块分别为选手1至选手4和选手5至选手8前3天的比赛日程。据此，将左 上角小块中的所有数字按其相对位置抄到右下角，又将左下角小块中的所有数字按其相对位置抄到右上角，这样我们就分别安排好了选手1至选手4和选手5至选手 8在后4天的比赛日程。依此思想容易将这个比赛日程表推广到具有任意多个选手的情形。

八、动态规划法

经常会遇到复杂问题不能简单地分解成几个子问题，而会分解出一系列的子问题。简单地采用把大问题分解成子问题，并综合子问题的解导出大问题的解的方法，问题求解耗时会按问题规模呈幂级数增加。

为了节约重复求相同子问题的时间，引入一个数组，不管它们是否对最终解有用，把所有子问题的解存于该数组中，这就是动态规划法所采用的基本方法。以下先用实例说明动态规划方法的使用。

【问题】 求两字符序列的最长公共字符子序列

问题描述：字符序列的子序列是指从给定字符序列中随意地（不一定连续）去掉若干个字符（可能一个也不去掉）后所形成的字符序列。令给定的字符序列X= “x0，x1，…，xm-1”，序列Y=“y0，y1，…，yk-1”是X的子序列，存在X的一个严格递增下标序列，使得对所有的j=0，1，…，k- 1，有xij=yj。例如，X=“ABCBDAB”，Y=“BCDB”是X的一个子序列。

给定两个序列A和B，称序列Z是A和B的公共子序列，是指Z同是A和B的子序列。问题要求已知两序列A和B的最长公共子序列。

如采用列举A的所有子序列，并一一检查其是否又是B的子序列，并随时记录所发现的子序列，最终求出最长公共子序列。这种方法因耗时太多而不可取。

考虑最长公共子序列问题如何分解成子问题，设A=“a0，a1，…，am-1”，B=“b0，b1，…，bm-1”，并Z=“z0，z1，…，zk-1”为它们的最长公共子序列。不难证明有以下性质：

（1） 如果am-1=bn-1，则zk-1=am-1=bn-1，且“z0，z1，…，zk-2”是“a0，a1，…，am-2”和“b0，b1，…，bn-2”的一个最长公共子序列；

（2） 如果am-1!=bn-1，则若zk-1!=am-1，蕴涵“z0，z1，…，zk-1”是“a0，a1，…，am-2”和“b0，b1，…，bn-1”的一个最长公共子序列；

（3） 如果am-1!=bn-1，则若zk-1!=bn-1，蕴涵“z0，z1，…，zk-1”是“a0，a1，…，am-1”和“b0，b1，…，bn-2”的一个最长公共子序列。

这样，在找A和B的公共子序列时，如有am-1=bn-1，则进一步解决一个子问题，找“a0，a1，…，am-2”和“b0，b1，…，bm-2” 的一个最长公共子序列；如果am-1!=bn-1，则要解决两个子问题，找出“a0，a1，…，am-2”和“b0，b1，…，bn-1”的一个最长公共 子序列和找出“a0，a1，…，am-1”和“b0，b1，…，bn-2”的一个最长公共子序列，再取两者中较长者作为A和B的最长公共子序列。

定义c[i][j]为序列“a0，a1，…，ai-2”和“b0，b1，…，bj-1”的最长公共子序列的长度，计算c[i][j]可递归地表述如下：

（1）c[i][j]=0 如果i=0或j=0；

（2）c[i][j]= c[i-1][j-1]+1 如果I，j>0，且a[i-1]=b[j-1]；

（3）c[i][j]=max（c[i][j-1]，c[i-1][j]） 如果I，j>0，且a[i-1]!=b[j-1]。

按此算式可写出计算两个序列的最长公共子序列的长度函数。由于c[i][j]的产生仅依赖于c[i-1][j-1]、c[i-1][j]和c[i][j-1]，故可以从c[m][n]开始，跟踪c[i][j]的产生过程，逆向构造出最长公共子序列。细节见程序。

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

|  |  |
| --- | --- |
| 01 | # include |
| 02 |  |
| 03 | # include |
| 04 |  |
| 05 | # define N 100 |
| 06 |  |
| 07 | **char** a[N],b[N],str[N]; |
| 08 |  |
| 09 | **int** lcs\_len(**char** \*a, **char** \*b, **int** c[ ][ N]) |
| 10 |  |
| 11 | { **int** m=strlen(a), n=strlen(b), i,j; |
| 12 |  |
| 13 | **for** (i=0;i<=m;i++) c[i][0]=0; |
| 14 |  |
| 15 | **for** (i=0;i<=n;i++) c[0][i]=0; |
| 16 |  |
| 17 | **for** (i=1;i<=m;i++) |
| 18 |  |
| 19 | **for** (j=1;j<=m;j++) |
| 20 |  |
| 21 | **if** (a[i-1]==b[j-1]) |
| 22 |  |
| 23 | c[i][j]=c[i-1][j-1]+1; |
| 24 |  |
| 25 | **else** **if** (c[i-1][j]>=c[i][j-1]) |
| 26 |  |
| 27 | c[i][j]=c[i-1][j]; |
| 28 |  |
| 29 | **else** |
| 30 |  |
| 31 | c[i][j]=c[i][j-1]; |
| 32 |  |
| 33 | **return** c[m][n]; |
| 34 |  |
| 35 | } |
| 36 |  |
| 37 | **char** \*buile\_lcs(**char** s[ ],**char** \*a, **char** \*b) |
| 38 |  |
| 39 | { **int** k, i=strlen(a), j=strlen(b); |
| 40 |  |
| 41 | k=lcs\_len(a,b,c); |
| 42 |  |
| 43 | s[k]='**\0**'; |
| 44 |  |
| 45 | **while** (k>0) |
| 46 |  |
| 47 | **if** (c[i][j]==c[i-1][j]) i--; |
| 48 |  |
| 49 | **else** **if** (c[i][j]==c[i][j-1]) j--; |
| 50 |  |
| 51 | **else** { s[--k]=a[i-1]; |
| 52 |  |
| 53 | i--; j--; |
| 54 |  |
| 55 | } |
| 56 |  |
| 57 | **return** s; |
| 58 |  |
| 59 | } |
| 60 |  |
| 61 | **void** main() |
| 62 |  |
| 63 | { printf (“Enter two **string**（<%d）!\n”,N); |
| 64 |  |
| 65 | scanf(“%s%s”,a,b); |
| 66 |  |
| 67 | printf(“LCS=%s\n”,build\_lcs(str,a,b)); |
| 68 |  |
| 69 | } |

1、动态规划的适用条件

任何思想方法都有一定的局限性，超出了特定条件，它就失去了作用。同样，动态规划也并不是万能的。适用动态规划的问题必须满足最优化原理和无后效性。

（1）最优化原理（最优子结构性质）

最优化原理可这样阐述：一个最优化策略具有这样的性质，不论过去状态和决策如何，对前面的决策所形成的状态而言，余下的诸决策必须构成最优策略。简而言之，一个最优化策略的子策略总是最优的。一个问题满足最优化原理又称其具有最优子结构性质。

例如图2中，若路线I和J是A到C的最优路径，则根据最优化原理，路线J必是从B到C的最优路线。这可用反证法证明：假设有另一路径J‘是B到C的最优路径，则A到C的路线取I和J’比I和J更优，矛盾。从而证明J‘必是B到C的最优路径。

最优化原理是动态规划的基础，任何问题，如果失去了最优化原理的支持，就不可能用动态规划方法计算。根据最优化原理导出的动态规划基本方程是解决一切动态规划问题的基本方法。

（2）无后向性

将各阶段按照一定的次序排列好之后，对于某个给定的阶段状态，它以前各阶段的状态无法直接影响它未来的决策，而只能通过当前的这个状态。换句话说，每个状态都是过去历史的一个完整总结。这就是无后向性，又称为无后效性。

（3）子问题的重叠性

动态规划算法的关键在于解决冗余，这是动态规划算法的根本目的。动态规划实质上是一种以空间换时间的技术，它在实现的过程中，不得不存储产生过程中的各种 状态，所以它的空间复杂度要大于其它的算法。选择动态规划算法是因为动态规划算法在空间上可以承受，而搜索算法在时间上却无法承受，所以我们舍空间而取时间。//Unity3D教程手册：www.unitymanual.com

所以，能够用动态规划解决的问题还有一个显着特征：子问题的重叠性。这个性质并不是动态规划适用的必要条件，但是如果该性质无法满足，动态规划算法同其他算法相比就不具备优势。

2、动态规划的基本思想

前文主要介绍了动态规划的一些理论依据，我们将前文所说的具有明显的阶段划分和状态转移方程的动态规划称为标准动态规划，这种标准动态规划是在研究多阶段 决策问题时推导出来的，具有严格的数学形式，适合用于理论上的分析。在实际应用中，许多问题的阶段划分并不明显，这时如果刻意地划分阶段法反而麻烦。一般 来说，只要该问题可以划分成规模更小的子问题，并且原问题的最优解中包含了子问题的最优解（即满足最优子化原理），则可以考虑用动态规划解决。

动态规划的实质是分治思想和解决冗余，因此，动态规划是一种将问题实例分解为更小的、相似的子问题，并存储子问题的解而避免计算重复的子问题，以解决最优化问题的算法策略。

由此可知，动态规划法与分治法和贪心法类似，它们都是将问题实例归纳为更小的、相似的子问题，并通过求解子问题产生一个全局最优解。其中贪心法的当前选择 可能要依赖已经作出的所有选择，但不依赖于有待于做出的选择和子问题。因此贪心法自顶向下，一步一步地作出贪心选择；而分治法中的各个子问题是独立的（即 不包含公共的子子问题），因此一旦递归地求出各子问题的解后，便可自下而上地将子问题的解合并成问题的解。但不足的是，如果当前选择可能要依赖子问题的解 时，则难以通过局部的贪心策略达到全局最优解；如果各子问题是不独立的，则分治法要做许多不必要的工作，重复地解公共的子问题。

解决上述问题的办法是利用动态规划。该方法主要应用于最优化问题，这类问题会有多种可能的解，每个解都有一个值，而动态规划找出其中最优（最大或最小）值 的解。若存在若干个取最优值的解的话，它只取其中的一个。在求解过程中，该方法也是通过求解局部子问题的解达到全局最优解，但与分治法和贪心法不同的是， 动态规划允许这些子问题不独立，（亦即各子问题可包含公共的子子问题）也允许其通过自身子问题的解作出选择，该方法对每一个子问题只解一次，并将结果保存 起来，避免每次碰到时都要重复计算。

因此，动态规划法所针对的问题有一个显着的特征，即它所对应的子问题树中的子问题呈现大量的重复。动态规划法的关键就在于，对于重复出现的子问题，只在第一次遇到时加以求解，并把答案保存起来，让以后再遇到时直接引用，不必重新求解。

3、动态规划算法的基本步骤

设计一个标准的动态规划算法，通常可按以下几个步骤进行：

（1）划分阶段：按照问题的时间或空间特征，把问题分为若干个阶段。注意这若干个阶段一定要是有序的或者是可排序的（即无后向性），否则问题就无法用动态规划求解。//Unity3D教程手册：www.unitymanual.com

（2）选择状态：将问题发展到各个阶段时所处于的各种客观情况用不同的状态表示出来。当然，状态的选择要满足无后效性。

（3）确定决策并写出状态转移方程：之所以把这两步放在一起，是因为决策和状态转移有着天然的联系，状态转移就是根据上一阶段的状态和决策来导出本阶段的 状态。所以，如果我们确定了决策，状态转移方程也就写出来了。但事实上，我们常常是反过来做，根据相邻两段的各状态之间的关系来确定决策。

（4）写出规划方程（包括边界条件）：动态规划的基本方程是规划方程的通用形式化表达式。

一般说来，只要阶段、状态、决策和状态转移确定了，这一步还是比较简单的。动态规划的主要难点在于理论上的设计，一旦设计完成，实现部分就会非常简单。根据动态规划的基本方程可以直接递归计算最优值，但是一般将其改为递推计算，实现的大体上的框架如下：

标准动态规划的基本框架

1. 对fn+1(xn+1)初始化; {边界条件}

for k:=n downto 1 do

for 每一个xk∈Xk do

for 每一个uk∈Uk(xk) do

begin

5. fk(xk):=一个极值; {∞或－∞}

6. xk+1:=Tk(xk,uk); {状态转移方程}

7. t:=φ(fk+1(xk+1),vk(xk,uk)); {基本方程(9)式}

if t比fk(xk)更优 then fk(xk):=t; {计算fk(xk)的最优值}

end;

9. t:=一个极值; {∞或－∞}

for 每一个x1∈X1 do

11. if f1(x1)比t更优 then t:=f1(x1); {按照10式求出最优指标}

12. 输出t;

但是，实际应用当中经常不显式地按照上面步骤设计动态规划，而是按以下几个步骤进行：

（1）分析最优解的性质，并刻划其结构特征。

（2）递归地定义最优值。

（3）以自底向上的方式或自顶向下的记忆化方法（备忘录法）计算出最优值。

（4）根据计算最优值时得到的信息，构造一个最优解。

步骤（1）～（3）是动态规划算法的基本步骤。在只需要求出最优值的情形，步骤（4）可以省略，若需要求出问题的一个最优解，则必须执行步骤（4）。此 时，在步骤（3）中计算最优值时，通常需记录更多的信息，以便在步骤（4）中，根据所记录的信息，快速地构造出一个最优解。

【问题】 凸多边形的最优三角剖分问题

问题描述：多边形是平面上一条分段线性的闭曲线。也就是说，多边形是由一系列首尾相接的直线段组成的。组成多边形的各直线段称为该多边形的边。多边形相接 两条边的连接点称为多边形的顶点。若多边形的边之间除了连接顶点外没有别的公共点，则称该多边形为简单多边形。一个简单多边形将平面分为3个部分：被包围 在多边形内的所有点构成了多边形的内部；多边形本身构成多边形的边界；而平面上其余的点构成了多边形的外部。当一个简单多边形及其内部构成一个闭凸集时， 称该简单多边形为凸多边形。也就是说凸多边形边界上或内部的任意两点所连成的直线段上所有的点均在该凸多边形的内部或边界上。

通常，用多边形顶点的逆时针序列来表示一个凸多边形，即P=表示具有n条边v0v1，v1v2，…，vn-1vn的一个凸多边形，其中，约定v0=vn 。

若vi与vj是多边形上不相邻的两个顶点，则线段vivj称为多边形的一条弦。弦将多边形分割成凸的两个子多边形和。多边形的三角剖分是一个将多边形分割成互不重迭的三角形的弦的集合T。图1是一个凸多边形的两个不同的三角剖分。

(a) (b)

图1 一个凸多边形的2个不同的三角剖分

在凸多边形P的一个三角剖分T中，各弦互不相交且弦数已达到最大，即P的任一不在T中的弦必与T中某一弦相交。在一个有n个顶点的凸多边形的三角刮分中，恰好有n-3条弦和n-2个三角形。

凸多边形最优三角剖分的问题是：给定一个凸多边形P=以及定义在由多边形的边和弦组成的三角形上的权函数ω。要求确定该凸多边形的一个三角剖分，使得该三角剖分对应的权即剖分中诸三角形上的权之和为最小。

可以定义三角形上各种各样的权函数ω。例如：定义ω(△vivjvk)=| vivj |+| vivk |+| vkvj |，其中，| vivj |是点vi到vj的欧氏距离。相应于此权函数的最优三角剖分即为最小弦长三角剖分。

（1）最优子结构性质

凸多边形的最优三角剖分问题有最优子结构性质。事实上，若凸（n+1）边形P=的一个最优三角剖分T包含三角形v0vkvn，1≤k≤n-1，则T的权为 3个部分权的和，即三角形v0vkvn的权，子多边形的权和的权之和。可以断言由T所确定的这两个子多边形的三角剖分也是最优的，因为若有或的更小权的三 角剖分，将会导致T不是最优三角剖分的矛盾。//Unity3D教程手册：www.unitymanual.com

（2）最优三角剖分对应的权的递归结构

首先，定义t[i，j]（1≤i的最优三角剖分所对应的权值，即最优值。为方便起见，设退化的多边形具有权值0。据此定义，要计算的凸（n+1）边多边形P对应的权的最优值为t[1，n]。

t[i，j]的值可以利用最优子结构性质递归地计算。由于退化的2顶点多边形的权值为0，所以t[i，i]=0，i=1，2，…，n 。当j一i≥1时，子多边形至少有3个顶点。由最优于结构性质，t[i，j]的值应为t[i，k]的值加上t[k+1，j]的值，再加上△vi- 1vkvj的权值，并在i≤k≤j-1的范围内取最小。由此，t[i，j]可递归地定义为：

（3）计算最优值

下面描述的计算凸（n+1）边形P=的三角剖分最优权值的动态规划算法MINIMUM\_WEIGHT，输入是凸多边形P=的权函数ω，输出是最优值t [i，j]和使得t[i，k]+t[k+1，j]+ω（△vi-1vkvj）达到最优的位置（k=）s[i，j]，1≤i≤j≤n 。

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

|  |  |
| --- | --- |
| 01 | Procedure MINIMUM\_WEIGHT(P，w)； |
| 02 |  |
| 03 | Begin |
| 04 |  |
| 05 | n=length[p]-1; |
| 06 |  |
| 07 | **for** i=1 to n **do** t[i,i]:=0; |
| 08 |  |
| 09 | **for** ll=2 to n **do** |
| 10 |  |
| 11 | **for** i=1 to n-ll+1 **do** |
| 12 |  |
| 13 | begin |
| 14 |  |
| 15 | j=i+ll-1; |
| 16 |  |
| 17 | t[i,j]=∞; |
| 18 |  |
| 19 | **for** k=i to j-1 **do** |
| 20 |  |
| 21 | begin |
| 22 |  |
| 23 | q=t[i,k]+t[k+1,j]+ω(△vi-1vkvj); |
| 24 |  |
| 25 | **if** qbegin |
| 26 |  |
| 27 | t[i,j]=q; |
| 28 |  |
| 29 | s[i,j]=k; |
| 30 |  |
| 31 | end; |
| 32 |  |
| 33 | end; |
| 34 |  |
| 35 | end; |
| 36 |  |
| 37 | **return**(t,s); |
| 38 |  |
| 39 | end; |

算法MINIMUM\_WEIGHT\_占用θ(n2)空间，耗时θ(n3)。

（4）构造最优三角剖分

如我们所看到的，对于任意的1≤i≤j≤n ，算法MINIMUM\_WEIGHT在计算每一个子多边形的最优三角剖分所对应的权值t[i，j]的同时，还在s[i，j]中记录了此最优三角剖分中与边 （或弦）vi-1vj构成的三角形的第三个顶点的位置。因此，利用最优子结构性质并借助于s[i，j]，1≤i≤j≤n ，凸（n+l）边形P=的最优三角剖分可容易地在Ο(n)时间内构造出来。

[Unity3D教程：游戏开发算法（一）](http://www.unitymanual.com/6784.html)

[Unity3D教程：游戏开发算法（二）](http://www.unitymanual.com/6787.html)

[Unity3D教程：游戏开发算法（三）](http://www.unitymanual.com/6821.html)