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：

```
{ 输入箱子的容积；
  输入物品种数 n；
  按体积从大到小顺序，输入各物品的体积；
  预置已用箱子链为空；
  预置已用箱子计数器 box_count 为 0；
  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；
  if （已用箱子都不能再放物品 i）
  { 另用一个箱子，并将物品 i 放入该箱子；
  box_count++；
  }
  else
  将物品 i 放入箱子 j；
  }
}
```

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】

```
# include
# include
typedef struct ele
{ int vno;
  struct ele *link;
} ELE;
typedef struct hnode
{ int remainder;
  ELE *head;
  Struct hnode *next;
} HNODE;

void main()
{ int n, i, box_count, box_volume, *a;
  HNODE *box_h, *box_t, *j;
  ELE *p, *q;
  Printf(“输入箱子容积\n”);
  Scanf(“%d”,&box_volume);
  Printf(“输入物品种数\n”);
  Scanf(“%d”,&n);
  A=(int *)malloc(sizeof(int)*n);
  Printf(“请按体积从大到小顺序输入各物品的体积：”);
  For (i=0;i          Box_h=box_t=NULL;
  Box_count=0;
  For (i=0;i          { p=(ELE *)malloc(sizeof(ELE));
  p->vno=i;
  for (j=box_h;j!=NULL;j=j->next)
  if (j->remainder>=a[i]) break;
  if (j==NULL)
  { j=(HNODE *)malloc(sizeof(HNODE));
  j->remainder=box_volume-a[i];
  j->head=NULL;
  if (box_h==NULL) box_h=box_t=j;
  else box_t=boix_t->next=j;
  j->next=NULL;
```

```
box_count++;
}
else j->remainder-=a[i];
for (q=j->next;q!=NULL&&q->link!=NULL;q=q->link);
if (q==NULL)
{ p->link=j->head;
j->head=p;
}
else
{ p->link=NULL;
q->link=p;
}
}
printf("共使用了%d 只箱子", box_count);
printf("各箱子装物品情况如下：");
for (j=box_h,i=1;j!=NULL;j=j->next,i++)
{ printf("第%2d 只箱子，还剩余容积%4d，所装物品有；\n",I,j->remainder);
for (p=j->head;p!=NULL;p=p->link)
printf("%4d",p->vno+1);
printf("\n");
}
}
```

【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】

```
# include
int delta_i[ ]={2,1,-1,-2,-2,-1,1,2};
int delta_j[ ]={1,2,2,1,-1,-2,-2,-1};
int board[8][8];
int exitn(int i,int j,int s,int a[ ])
{ int i1,j1,k,count;
  for (count=k=0;k<8;k++)
  { i1=i+delta_i[(s+k)%8];
  j1=i+delta_j[(s+k)%8];
  if (i1>=0&&i1<8&&j1>=0&&j1<8&&board[I1][j1]==0)
  a[count++]=(s+k)%8;
  }
  return count;
}

int next(int i,int j,int s)
{ int m,k,mm,min,a[8],b[8],temp;
  m=exitn(i,j,s,a);
  if (m==0) return －1;
  for (min=9,k=0;k      { temp=exitn(I+delta_i[a[k]],j+delta_j[a[k]],s,b);
  if (temp      { min=temp;
kk=a[k];
  }
  }
  return kk;
}

void main()
{ int sx,sy,i,j,step,no,start;
  for (sx=0;sx<8;sx++)
  for (sy=0;sy<8;sy++)
  { start=0;
  do {
  for (i=0;i<8;i++)
  for (j=0;j<8;j++)
  board[i][j]=0;
  board[sx][sy]=1;
```

```
  I=sx; j=sy;
  For (step=2;step<64;step++)
  { if ((no=next(i,j,start))==-1) break;
  I+=delta_i[no];
  j+=delta_j[no];
  board[i][j]=step;
  }
  if (step>64) break;
  start++;
  } while(step<=64)
  for (i=0;i<8;i++)
  { for (j=0;j<8;j++)
  printf( “%4d” ,board[i][j]);
  printf( “\n\n” );
  }
  scanf( “%*c” );
  }
}
```