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：

fib(0)=0;

fib(1)=1;

fib(n)=fib(n-1)+fib(n-2) （当 n>1 时）。

写成递归函数有：

```
int fib(int n)
{ if (n==0) return 0;
  if (n==1) return 1;
  if (n>1) return fib(n-1)+fib(n-2);
}
```

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】

```
# include
# define MAXN 100
int a[MAXN];
void comb(int m,int k)
{ int i,j;
 for (i=m;i>=k;i--)
 { a[k]=i;
 if (k>1)
 comb(i-1,k-1);
 else
 { for (j=a[0];j>0;j--)
 printf(“%4d”,a[j]);
 printf(“\n”);
 }
 }
}

void main()
{ a[0]=3;
 comb(5,3);
}
```

【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：

```
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)
{ /*考虑物品 i 包含在当前方案中的可能性*/
 if(包含物品 i 是可以接受的)
 { 将物品 i 包含在当前方案中；
 if (i        try(i+1,tw+物品 i 的重量,tv);
 else
 /*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*/
以当前方案作为临时最佳方案保存;
 恢复物品 i 不包含状态；
 }
 /*考虑物品 i 不包含在当前方案中的可能性*/
 if (不包含物品 i 仅是可男考虑的)
 if (i        try(i+1,tw,tv-物品 i 的价值)；
 else
 /*又一个完整方案，因它比前面的方案好，以它作为最佳方案*/
以当前方案作为临时最佳方案保存;
 }
```

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】

```
# include
# define N 100
double limitW,totV,maxV;
int option[N],cop[N];
struct { double weight;
 double value;
 }a[N];
int n;
void find(int i,double tw,double tv)
{ int k;
 /*考虑物品 i 包含在当前方案中的可能性*/
 if (tw+a[i].weight<=limitW)
 { cop[i]=1;
```

```
  if (i        else
  { for (k=0;k        option[k]=cop[k];
  maxv=tv;
   }
  cop[i]=0;
}
  /*考虑物品 i 不包含在当前方案中的可能性*/
  if (tv-a[i].value>maxV)
  if (i        else
  { for (k=0;k        option[k]=cop[k];
  maxv=tv-a[i].value;
   }
}

void main()
{ int k;
  double w,v;
  printf("输入物品种数\n");
  scanf(("%d",&n);
  printf("输入各物品的重量和价值\n");
  for (totv=0.0,k=0;k        { scanf("%1f%1f",&w,&v);
  a[k].weight=w;
  a[k].value=v;
  totV+=V;
   }
  printf("输入限制重量\n");
  scanf("%1f",&limitV);
  maxv=0.0;
  for (k=0;k        find(0,0.0,totV);
  for (k=0;k        if (option[k]) printf("%4d",k+1);
  printf("\n 总价值为%.2f\n",maxv);
}
```

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】

```
# include
# define N 100
double limitW;
```

```
int cop[N];
struct ele { double weight;
  double value;
  } a[N];
int k,n;
struct { int flg;
  double tw;
  double tv;
  }twv[N];
void next(int i,double tw,double tv)
{ twv[i].flg=1;
  twv[i].tw=tw;
  twv[i].tv=tv;
}
double find(struct ele *a,int n)
{ int i,k,f;
  double maxv,tw,tv,totv;
  maxv=0;
  for (totv=0.0,k=0;k          totv+=a[k].value;
  next(0,0.0,totv);
  i=0;
  While (i>=0)
  { f=twv[i].flg;
  tw=twv[i].tw;
  tv=twv[i].tv;
  switch(f)
  { case 1: twv[i].flg++;
  if (tw+a[i].weight<=limitW)
  if (i          { next(i+1,tw+a[i].weight,tv);
  i++;
  }
  else
  { maxv=tv;
  for (k=0;k          cop[k]=twv[k].flg!=0;
  }
  break;
  case 0: i--;
  break;
  default: twv[i].flg=0;
  if (tv-a[i].value>maxv)
  if (i          { next(i+1,tw,tv-a[i].value);
  i++;
  }
  else
```

```
 { maxv=tv-a[i].value;
 for (k=0;k       cop[k]=twv[k].flg!=0;
 }
 break;
 }
 }
 return maxv;
}

void main()
{ double maxv;
 printf(“输入物品种数\n”);
 scanf((“%d”,&n);
 printf(“输入限制重量\n”);
 scanf(“%1f”,&limitW);
printf(“输入各物品的重量和价值\n”);
 for (k=0;k       scanf(“%1f%1f”,&a[k].weight,&a[k].value);
 maxv=find(a,n);
 printf(“\n 选中的物品为\n”);
for (k=0;k       if (option[k]) printf(“%4d”,k+1);
 printf(“\n 总价值为%.2f\n”,maxv);
}
```