递归的概念

- 递归算法：一个直接或间接地调用自身的算法
- 递归函数：使用函数自身给出定义的函数
- 递归方程：对于递归算法，一般可把时间代价表示为一个递归方程
- 解递归方程最常用的方法是进行递归扩展 **阶乘函数**
    1. 边界条件
    2. 递归关系

$n! = \begin{cases} 1, n = 1 \\ n(n-1)!, n > 1 \end{cases}$

Fibonacci数列 $F(n) = \begin{cases} 1, n = 0 \\ 1, n = 1 \\ F(n-1) + F(n-2), n > 1 \end{cases}$

==初始条件==和==递归方程==是递归函数的两个要素

Ackerman函数 $\begin{cases} A(1,0) = 2 \\ A(0,m) = 1, m >= 0 \\ A(n,0) = n+2, n >= 2 \\ A(n,m) = A(A(n-1,m), m-1), n、m >= 1 \end{cases}$

- Ackerman函数为**双递归函数**
- 双递归函数：一个**函数**及它的**一个变量**由函数自身定义
- A(n,m)的自变量m的每一个值都定义了一个单变量函数

注：需推一遍证明

排列问题

设R={r_1,r_2,...,r_n}是要进行排列的n个元素，R_i=R-{r_i}。集合X中的元素的全排列记为Perm(X)。

（r_i)Perm(X)表示在全排列Perm(X)的每个排列前加上前缀r_i得到的排列。

R的全排列定义如下：

当n=1时，Perm(R)=(r)，其中r是集合R中唯一的元素；

当n>1时，Perm(R)由(r_1)Perm(R_1)，(r_2)Perm(R_2)，…，(r_n)Perm(R_n)构成。

为便于理解，以{1,2,3,4,5,6}为例：

1 2 3 4 5 6 -> **1** 2 3 4 5 6 -> **1 2** 3 4 5 6 -> **1 2 3** 4 5 6 -> **1 2 3 4** 5 6 -> **1 2 3 4 5** 6

-> **1 2 3 4** 6 5 -> **1 2 3 4 6** 5

```
template<class Type>
void Perm(Type list[],int k, int m){             //产生list[k:m]的所有排列
    if(k==m){                                    //只剩下一个元素，到达递归的最底层
        for (int i=0;i<=m;i++)
            cout << list[i];
        cout << endl;
    }
    else{                                        //还有多个元素待排列，递归产生排列
        for (int i=k;i<=m;i++)
        {
            Swap(list[k],list[i]);
            Perm(list,k+1,m);
            Swap(list[k],list[i]);               //复位，保证所有元素都能依次做前缀
```

```
        }
    }
}

template<class Type>
inline void Swap(Type &a, Type &b)
{
    Type temp=a;
    a = b;
    b = temp;
}
```

整数划分问题

将正整数n表示成一系列正整数之和，$n=n_1+n_2+\ldots+n_k(n_1\geq n_2\geq\ldots\geq n_k，k\geq 1)$。

正整数n的不同的划分个数称为正整数n的划分数，记为p(n)。以正整数6为例：

6；

5+1；

4+2；4+1+1；

3+3；3+2+1；3+1+1+1；

2+2+2；2+2+1+1；2+1+1+1+1；

1+1+1+1+1+1。

将最大加数n_1不大于m的划分个数记作q(n,m)，建立如下递归关系：

$$q(n.m)=\begin{cases}1,n=1、m=1\\q(n,n),n<m\\1+q(n,n-1),n=m\\q(n,m-1)+q(n-m,m),n>m>1\end{cases}$$

1. q(n,1)=1，n≥1。当最大加数n_1不大于1时，任何正整数n只有一种划分形式，即n=1+1+...+1
2. q(n,m)=q(n,n)，m≥n。最大加数n_1不能大于n。
3. q(n,n)=1+q(n,n-1)。正整数n的划分由n_1=n的划分和n_1≤n-1的划分组成；n_1=n时，划分仅有一种。
4. q(n,m-1)+q(n-m,m)，n>m>1。正整数n的最大加数n_1不大于m的划分由n_1=m的划分和n_1≤m-1的划分组成。

Hanoi塔问题

设a，b，c是3个塔座。开始时，在塔座a上有一叠共n个圆盘，这些圆盘自下而上，由大到小地叠在一起。各圆盘从小到大编号为1，2，...，n，现要求将塔座a上的这一叠圆盘移到塔座b上，并仍按同样顺序叠置。在移动圆盘时遵守：

1. 每次只移动1个圆盘
2. 任何时刻都不允许将较大的圆盘压到较小的圆盘之上
3. 在满足1和2的前提下，可将圆盘移动至a，b，c中任一塔座上

$$T(n)=\begin{cases}2T(n-1)+1,n>1\\1,n=1\end{cases}$$

```
void hanoi(int n,a,b,c)             //a上n个盘借助c移到b
{
    if (n==1)                       //将第1个盘从a移到b
        move(1,a,b);
    else{
        hanoi(n-1,a,c,b);           //将第1至n-1个盘，从a移到c
        move(n,a,b);                //将第n个盘从a移到b
        hanoi(n-1,c,b,a);           //将第1至n-1个盘从c移到b
    }
}
```

使用这种递归调用的关键在于建立**递归调用工作栈**。

递归程序逐层调用需要分配存储空间，一旦某一层被启用，就要为之开辟新的空间。当一层执行完毕，释放相应空间，退到上一层。

递归优点：结构清晰，可读性强

递归缺点：递归算法的运行效率较低，无论是耗费的计算时间还是占用的存储空间都比非递归算法要多

解决方法：在递归算法中消除递归调用，使其转化为非递归算法。可采用一个用户定义的栈来模拟系统的递归调用工作栈。

分治策略

基本思想

将问题分解成若干个子问题，然后求解子问题。分治策略可以递归地进行，即子问题仍然可以用分值策略来处理，但最后的问题要非常基本而简单。

```
divide-and-conquer(P){
    if (|P| <= n0)                                  //直接解决小规模问题
        adhoc(P);
    divide P into smaller subinstances P1,P2,…,Pk   //分解问题
    for (i = 1; i<=k; i++)                          //递归解各子问题
        yi=divide-and-conquer(Pi);                  //将格子问题的解合并为原问题的解
    return merge(y1,y2,…,yk)
}
```

代价分析 $\begin{cases} O(1), n=1 \\ kT(n/m)+f(n), n>1 \end{cases}$

$$T(N)=aT(N/b)+N^k, a\geq 1, b>1 \; T(n)=\begin{cases} O(N^t), t=log_b a, a>b^k \\ O(N^k logN), a=b^k \\ O(N^k), a<b^k \end{cases}$$

二分搜索技术

给定已按升序排好序的n个元素a[0:n-1]，现要在这n个元素中找出一特定元素x。

1. 顺序搜索方法：最好时1次，最坏时n次
2. 二分搜索方法

   基本思想

   将n个元素分成个数大致相同的两半，取a[n/2]与x作比较。

   - 如果x=a[n/2]，则找到x，算法终止；
   - 如果x<a[n/2]，则在数组a的左半部继续搜索x；
   - 如果x>a[n/2]，则在数组a的右半部继续搜索x。

```
template<class Type>
int BinarySearch(Type a[],const Type& x, int l, int r)  //l=0,r=n-1
{//找到x时返回其在数组中的位置，否则返回-1
    while(r >= l){
        int m = (l+r)/2;
        if(x==a[m])
            return m;
        if (x<a[m])
            r=m-1;
        else
            l=m+1;
    }
    return -1;
}
```

$T（n）=\begin{cases}T(n/2)+O(1),n>1\\O(1),n=1\end{cases}$

求解：$T(n)=logn$

复杂性：$O(logn)$

大整数的乘法

设X和Y都是n位二进制整数，计算它们的乘积XY。

小学方法：$O(n^2)$

将n位二进制整数X和Y分为2段，每段的长为n/2位，由此$X=A\times2^t+B,t=n/2$，$Y=C\times2^t+D,t=n/2$

由此，$XY=(A\times2^t+B)(C\times2^t+D)=AC\times2^t+(AD+BC)\times2^t+BD,t=n/2$

分治法：$XY=AC\times2^n+((A-B)(D-C)+AC+BD)\times2^t+BD,t=n/2$

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。

$ T（n）=\begin{cases}O(1),n=1\\3T(n/2)+O(n),n>1\end{cases}$ 易求得，其解为 $T(n)=O(n^t),t=log3\approx1.59$

Strassen矩阵乘法

设A和B是两个$n\times n$矩阵，乘积AB同样是一个$n\times n$矩阵，A和B乘积矩阵C中元素定$c[i][j]$定义为 (C[i][j]=\sum_{k=1}^{n}{A[i][k]B[k][j]}) 传统方法：$O(n^3)$

分治法：

$[C[1][1] \quad C[1][2]\mathbb{C}[2][1] \quad C[2][2]]=[A[1][1] \quad A[1][2]\textcolor{red}{\backslash A}[2][1] \quad A[2][2]]\,[B[1][1] \quad B[1][2]\textcolor{red}{\backslash B}[2][1] \quad B[2][2]]$

由此可得

$C[1][1]=A[1][1]B[1][1]+A[1][2]B[2][1]\;C[1][2]=A[1][1]B[1][2]+A[1][2]B[2][2]$
$C[2][1]=A[2][1]B[1][1]+A[2][2]B[2][1]\;C[2][2]=A[2][1]B[1][2]+A[2][2]B[2][2]$

根据上述算法可得，计算2个n阶方阵的乘积转化为计算8个n/2阶方阵的乘积和4个n/2阶方阵的加法。

故分治法的计算时间耗费$T(n)$应满足$T（n）=\begin{cases}O(1),n=2\\8T(n/2)+O(n^2),n>2\end{cases}T(n)=O(n^3)$

改进后变成了7次乘法（*详情见书上P20*），此时计算时间耗费$T(n)$满足

$T（n）=\begin{cases}O(1),n=2\\7T(n/2)+O(n^2),n>2\end{cases}$解此递归方程得$T(n)=O(n^t),t=log7\approx2.81$

棋盘覆盖

在一个$2^k \times 2^k$个方格组成的棋盘中，恰有一个方格与其它方格不同，称为特殊方格，且称该棋盘为一特殊棋盘。在该问题中，要用图示的4中不同形态的L型骨牌覆盖给定的特殊棋盘上除特殊方格以外的所有方格，且任何2个L型骨牌不得重叠覆盖。

棋盘覆盖

当k>0时，将$2^k \times 2^k$个棋盘分割为4个2^k-1^×2^k-1^子棋盘。特殊方格必定位于4个较小子棋盘之一中，其余3个子棋盘中无特殊方格。为了将这3个无特殊方格的子棋盘转化为特殊棋盘，用一个L型骨牌覆盖这3个较小棋盘的会合处，将原问题转化为4个较小规模的棋盘覆盖问题，递归地使用这种分割，直至棋盘简化为1×1。

棋盘覆盖2

```
/*
tr:棋盘左上角方格的行号              tc:棋盘左上角方格的列号
dr:特殊方格所在的行号                dc:特殊方格所在的列号
size：size=2^k，棋盘规格为2^k×2^k
*/
void ChessBoard(int tr,int tc,int dr,int dc,int size)
{
    if (size==1) return;
    int t = tile++;                             //L型骨牌号
    s = size/2;                                 //分割棋盘
    //覆盖左上角子棋盘
    if (dr<tr+s && dc<tc+s)                     //特殊方格在此棋盘中
        ChessBoard(tr,tc,dr,dc,s);
    else{                                       //此棋盘无特殊方格
        Board[tr+s-1][tc+s-1]=t;                //用t号L型骨牌覆盖右下角
        ChessBoard(tr,tc,tr+s-1,tc+s-1,s);      //覆盖其余方格
    }
    //覆盖右上角子棋盘
    if (dr<tr+s && dc>=tc+s)                    //特殊方格在此棋盘中
        ChessBoard(tr,tc+s,dr,dc,s);
    else{                                       //此棋盘中无特殊方格
        Board[tr+s-1][tc+s]=t;                  //用t号L型骨牌覆盖左下角
        ChessBoard(tr,tc+s,tr+s-1,tc+s,s);      //覆盖其余方格
    }
    //覆盖左下角子棋盘
    if (dr>=tr+s && dc<tc+s)
        ChessBoard(tr+s,tc,dr,dc,s);
    else{
        Board[tr+s][tc+s-1]=t;
        ChessBoard(tr+s.tc,tr+s,tc+s-1,s);
    }
    //覆盖右下角子棋盘
    if (dr>=tr+s && dc>=tc+s)
        ChessBoard(tr+s,tc+s,dr,dc,s);
    else{
        Board[tr+s][tc+s]=t;
        ChessBoard(tr+s,tc+s,tr+s,tc+s,s);
    }
}
```

由此可得该时间复杂度的递推公式为$T(n) = \begin{cases} 4T(n/2) + O(1), n > 1 \\ O(1), n = 1 \end{cases}$解得$T(n) = O(n^2) = O(4^k)$

合并排序

- 合并是将两个或多个有序表合并成一个有序表
- 二路合并：对两个已排好序的表进行合并
- 合并排序算法是用分治策略实现对n个元素进行排序的算法

基本思想

将待排序元素分成大小大致相同的两个子集合，分别对两个子集合进行排序，最终将排好序的子集合合并成要求的排好序的集合。

将待排序集合一分为二，直至待排序集合只剩下一个元素为止，然后不断合并两个排好序的数组段。

```
template<class Type>
void MergeSort(Type a[],int left,int right)
{
    if (left<right){                        //至少有2个元素
        int i = (left+right)/2;             //取中点
        MergeSort(a, left, i);
        MergeSort(a, i+1, right);
        Merge(a, b, left, i, right);        //合并到数组b
        Copy(a,b,left,right);               //复制回数组a
    }
}
```

在最坏情况下所需的计算时间$T(n)$满足 $T（n）=\begin{cases}O(1),n\le 1\\2T(n/2)+O(n),n>1\end{cases}$ 解此递归方程可得 $T(n)=O(nlogn)$

两组归并算法

作用：将两组有序文件合并成一组有序文件。

```
void Merge(Type c[],Type d[], int l, int m, int r)
{
    //c[l]到c[m]、c[m+1]到c[r]是两有序文件合并到d
    int i,j,k;
    i = l;
    j = m+1;
    k = l;
    while((i<=m)&&(j<=r)){
        if (c[i]<=c[j])         //从两个有序文件中的第一个记录中选出小的记录
            d[k++]=c[i++];
        else
            d[k++]=c[j++]
    }
    while(i<=m)                 //复制第一个文件的剩余记录
        d[k++]=c[i++];
    while(j<=r)                 //复制第二个文件的剩余记录
        d[k++]=c[j++];
}
```

无递归形式算法

```
void MergeSort(Type a[], int n)
{
    Type *b = new Type[n];
    int s = 1;
```

```
    while (s<n){
        MergePass(a,b,s,n);     //一趟归并结果放在b中
        s*=2;
        MergePass(b,a,s,n);     //一趟归并结果放在a中
        s+=s;
    }
}

//调用两组归并算法，对数组进行一趟归并，将长为n的有序文件归并成长为2n的有序数组
void MergePass(Type x[], Type y[], int s, int n)
{//对x做一趟归并，结果放在y中
    int i = 0,j;                            //n为本趟归并的有序子数组的长度
    while(i+2*s-1<n)
    {
        Merge(x,y,i,i+s-1,i+2*s-1);     //归并长度为s的两子数组
        i = i+2*s;
    }
    if (i+s<n)                          //剩下两个子数组，其中一个长度小于s
        Merge(x,y,i,i+s-1,n-1);
    else
        for (j=i;j<n;j++)               //将最后一个子数组复制到数组y中
            y[j]=x[j];
}
```

最好情况：$T(n)=2T(n/2)+n/2\ T(1)=0\ T(2)=1\ T(4)=4$

最坏情况：$T(n)=2T(n/2)+n-1\ T(1)=0\ T(2)=1\ T(4)=5$

最坏时间复杂度：$O(nlogn)$

最好时间复杂度：$O(nlogn)$

平均时间复杂度：$O(nlogn)$

辅助空间：$O(n)$

快速排序

基本思想

1. 设定基准值

   使序列中的每个元素与基准值比较

2. 基于基准值划分子序列

   序列中比基准值小的放在基准值的左边，形成左部；大的放在右边，形成右部。

3. 递归执行

   左部和右部分别递归地执行上面的过程：选基准值，小的放在左边，大的放在右边，直到排序结束。

```
/*
p:序列的左边界（最左元素下标）
r:序列的右边界（最右元素下标）
q:下一次左右两边子序列的分区位置
*/
template<class Type>
void QuickSort(Type a[], int p, int r)
{
    if (p<r){
        int q = Partition(a,p,r);
        QuickSort(a,p,q-1);             //对左半段排序
```

```
        QuickSort(a,q+1,r);              //对右半段排序
    }
}

template<class Type>
int Partition(Type a[],int p,int r)
{
    int i = p, j = r+1;
    Type x=a[p];                //设定基准值
    //将小于x的元素交换到左边区域，将大于x的元素交换到右边区域
    while(true){
        while(a[++i]<x);
        while(a[--j]>x);
        if (i>=j)
            break;
        Swap(a[i],a[j]);
    }
    a[p]=a[j];
    a[j]=x;
    return j;
}
```

快速排序

最坏时间复杂度：$O(n^2)$

最好/平均时间复杂度：$O(nlog_2n)$

最坏辅助空间：$O(n)$

最好/平均辅助空间：$O(log_2n)$

最坏情况

序列的n个元素已经排好序，每次划分都恰好把序列分为1，n-1两部分，那么总共需要n-1次划分，每次比较的次数分别为n-1，n-2，...，1，所以整个比较次数约为$n(n-1)/2$~$n^2/2$ ($T(n)=\begin{cases}O(1),n\le 1\\T(n-1)+O(n),n>1\end{cases}$ 求解递归方程可得T(n)=O(n^2)，相应的，$n-1$次划分形成的子序列共需要O(n)辅助空间

最好情况

每次划分所取的基准都恰好为序列中值，可将序列从中间均衡划分为等长子序列，因此需要$\log_2$n次划分即可完成排序。$T(n)=\begin{cases}O(1),n\le 1\\2T(n/2)+O(n),n>1\end{cases}$ 求解递归方程可得$T(n)=O(nlog_2n)$，形成的子序列共需要$O(log_2n)$辅助空间。

平均情况

设分划的数x=a[p]在最后排序的第k位，第1轮比较n-1次，前有k-1个元素，后有n-k个元素。

==快速排序是不稳定的排序算法==

通过修改算法partition，可以设计采用随机选择策略的快速排序算法，在快速排序算法的每一次partition中，可在a[p:r]中随机选出一个元素作为划分基准，从概率角度使得划分的期望达到对称效果。

```
template<class Type>
int RandomizedPartition(Type a[],int p, int r)
{
    int i = Random(p,r);
    Swap(a[i],a[p]);
    return Partition(a,p,r);
}
```

线性时间选择

给定线性序集中n个元素和一个整数k（1≤k≤n），要求找出这n个元素中第k小的元素，即如果将这个元素依其线性序排列时，排在第k个位置的元素即为要找的元素。

复杂性是O(n^2^)随机化算法

```
template<class Type>
Type RandomizedSelect(Type a[],int p, int r, int k)
{
    if (p==r)
        return a[p];
    int i = RandomizedPartition(a,p,r);
    j = i-p+1;
    if(k<=j)
        return RandomizedSelect(a,p,i,k);
    else
        return RandomizedSelect(a,i+1,r,k-j);
}
```

算法时间复杂性：$O(n)$

按以下步骤找到满足要求的划分标准：

1. 将n个输入元素划分成[n/5]个组，每组5个元素，除可能有一个组不是5个元素外。用任意一种排序算法，将每组中的元素排好序，并取出每组的中位数，共[n/5]个。
2. 递归调用Select找出这[n/5]个元素的中位数。如果[n/5]是偶数，就找它的两个中位数中较大的一个。

```
template<class Type>
Type Select(Type a[], int p, int r, int k)
{
    if (r-p<cn)
        用简单的排序算法对数组a[p:r]排序;
        return a[p+k-1];
    for (int i = 0; i <= (r-p-4)/5;i++)
    {
        //将a[p+5*i]至a[p+5*i+4]的第3小元素与a[p+i]交换位置
        Type x = Select(a,p,p+(r-p-4)/5,(r-p-4)/10);
        int i = Partition(a,p,r,x);
        if (k<=j)
            return Select(a,p,i,k);
        else
            return Select(a,i+1,r,k-j);
    }
}
```

关于$T(n)$的递归式$T(n) \leq \begin{cases} T(\frac{1}{5}n)+T(\frac{3}{4}n)+cn, n>>5 \\ cn, n \leq 5 \end{cases}$

最接近点对问题

给定平面上n个点，找其中的一对点，使得在n个点组成的所有点对中，该点对间的距离最小。

将所给的平面上n个点的集合S分成两个子集S_1和S_2，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。

一维点集S

```
bool Cpair(S,d){
    n = |S|;
    if (n<2){
        d=∞;
        return false;
    }
    m = S中各点坐标的中位数;
    构造S1和S2;                    //S1={x∈S|x<=m},S2={x∈S|x>m}
    Cpair(S1,d1);
    Cpair(S2,d2);
    p=max(S1);
    q=min(S2);
    d=min(d1,d2,q-p);
    return true;
}
```

算法耗费的计算时间$T(n)$满足递归方程 $T(n)=\begin{cases}2T(n/2)+O(n), n>4\\O(1), n=4\end{cases}$ 求得$T(n)=O(nlogn)$

二维点集

S={p_1,p_2,...,p_n}

作一条垂直线$l:x=m$将平面分为两部分，设(p,q)为S的最接近点对情况：

- p,q∈S_1
- p,q∈S_2
- p∈S_1，q∈S_2
    - p∈S_1，q∈S_2的情形
    - P_1、P_2有稀疏 性质：对P_1中任意一点p，在P_2中最多只有6个S中的点使与p的距离不超过d（*鸽舍原理*)

```
bool Cpair2(S,d)
{
    n=|S|;
    if (n<2){d=∞;return false;}
    m=S中各点x间坐标的中位数;
    构造S1和S2;
    //S1={x∈S|x<=m},S2={x∈S|x>m}
    Cpair2(S1,d1);
    Cpair2(S2,d2);
    dm = min(d1,d2);
    设P1是S1中距垂直分割线l的距离在dm之内的所有点组成的集合;
    P2是S2中距分割线l的距离在dm之内所有点组成的集合;
    将P1和P2中点依其y坐标值排序;
    并设X和Y是相应的已排好序的点列;
    通过扫描X以及对于X中每个点检查Y中与其距离在dm之内的所有点(最多6个)可以完成合并;
    当X中的扫描指针逐次向上移动时，Y中的扫描指针可在宽为2dm的一个区间内移动;
```

```
    设dl是按这种扫描方式找到的点对间的最小距离;
    d=min(dm,dl);
    return true;
}
```

计算时间$T(n)$满足递归方程 $T(n)=\begin{cases}2T(n/2)+O(n), n>4\\O(1), n\le 4\end{cases}$可解得$T(n)=O(nlogn)$

循环赛日程表

设有$n=2^k$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. 每个选手一天只能赛一次
3. 循环赛一共进行n-1天

分治策略：

将所有选手对分为两组，n个选手的比赛日程表可通过为n/2个选手设计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分割，直到只剩下2个选手时，比赛日程表的制定就变得很简单，只需让这2个选手进行比赛即可。

```
void Table(int k, int **a)
{
    int n = 1;
    for (int i=1;i<=k;i++)
        n*=2;
    for (int i=1;i<=n;i++)
        a[1][i]=i;
    int m = 1;
    for (int s=1;s<=k;s++){
        for (int i=m+1;i<=2*m;i++){
            for (int j=m+1;j<=2*m;j++)
            {
                a[i][j+(t-1)*m*2]=a[i-m][j+(t-1)*m*2-m];
                a[i][j+(t-1)*m*2-m]=a[i-m][j+(t-1)*m*2];
            }
        }
        m*=2;
    }
}
```