数据结构与算法

绪论

基本术语

数据分为**数值性数据**和**非数值数据**，其基本单位是**数据元素**，是计算机处理或访问的基本单位；

一个数据元素可以由若干数据项组成，**数据项**又称**属性，字段，域，分为初等项**（不可分割）和**组合项**；

结构：数据元素之间的关系；

数据结构是由与特定问题相关的某一数据元素的集合(*对象*)(D)和该集合中数据元素之间的关系(R)组成的；分为**静态数据结构**和**动态数据结构**，舍弃了实际的物理背景，是通用型的定义； 数据结构={D,R}

数据对象：**狭义：具有一定关系的*相同性质*的数据元素的集合**；**广义：由数据抽象和处理数据构成的封装体**

数据类型：**一个值的集合和定义在这个值集合上的一组操作的总称**，分为**内置数据类型（亦称基本数据类型或原子类型，可直接使用）和构造数据类型（由不同成分的内置数据类型子结构按照一定的规则组成，是用编程语言描述的数据结构的存储映像）**，是**数据结构的实例化**；数据类型**再实例化**得到具体变量，如**类与对象**；

抽象数据类型ADT（Abstract Data Type)：

抽象的本质就是抽取反映问题本质的东西，忽略非本质的细节；

特性：**使用与实现相分离**，**数据*封装*与*信息*隐蔽**；

在C++，java中用类描述。C中一般不使用ADT，数据和结构是分开的。

数据结构：

一，**分解和抽象**：1，**数据分解划分出数据的层次，再抽象得到数据的*逻辑结构***；2，**处理分解划分成各种功能，再通过抽象得到算法的定义**。这是一个从具体（具体问题）到抽象（数据结构与算法）的过程；进一步**通过对实现细节的进一步考虑得到存储结构和实现运算，从而完成程序设计的任务**，实现从数据结构到具体实现。

二，逻辑结构和存储结构

数据的逻辑结构根据问题所要实现的功能建立（不考虑具体实现），存储结构根据问题所要的需求（响应速度，处理时间，等）来实现数据的逻辑结构。（数据结构一般指的就是逻辑结构，逻辑结构相同，即使存储结构不同也是相同的逻辑结构）；

逻辑结构的分类：

线性结构，树形结构，图结构，集合结构。

线性结构：元素之间的关系是一**对**一的，如线性表，向量，栈，队列，优先队列，字典等；

非线性结构：每个数据结构可能**与零个或多个其他数据元素发生联系**，分为树结构（一对多）和图结构（多对多）。 如多维数组和广义表等；

集合结构的实现往往采用其他逻辑结构的存储表示。

数据结构的存储结构：

存取结构根据存取方法的不同分为三类：

1，**直接存取结构**（向量，多维数组，散列表）；

2，**顺序存取结构**（各种链表，图的邻接表）；

3，**索引存取结构**（线性索引，多叉查找树）；

常用的四种存储结构

1，**顺序存储**：元素之间的逻辑关系由**存储单元的邻接位置关系体现**，由此得到顺序存储结构，借助**一维数组**描述；

2，**链接存储**：元素之间的逻辑关系由**附加的链接指针指示**，由此得到链表存储结构，借助**指针类型**描述；

3，**索引存储**：在存储元素信息的同时还建立**索引表**，其中每一项称为**索引项（包括关键码和地址）**，按针对一个元素还是一组元素分为**稠密索引和稀疏索引**；按是一层还是多层分为线**性索引和多级索引**；

4，**散列存储**：根据元素的关键码**通过一个函数计算得到**元素的存储地址。

前两种在**内存**中，也是基本的**两种物理存储结构**。后两个在**外存**中。

选择存储结构的要素有<1>*访问频率*，<2>*修改频率*，<3>*安全保密*；

定义在数据结构上的操作：

1，创建；2，销毁；3，查找；4，插入；5，删除；6，排序。

好的数据结构：**可以通过某种“线性化”规则被转化为线性结构，通常对应好的算法**；

计算机本质上**只能按照逻辑顺序处理指令和内存单元**，例如图，树的遍历查找需要线性化。

C语言知识点补充

二维数组传参

不常用

```
不用区分行列遍历
 void Print(int *p,int n)//或int p[];
 {
     for(int i=0;i<n;i++)//存储在一块连续的区域
     printf("%d ",p[i]);
printf("\n");
 }

int main()
{
    int s[2][3]={1,2,3,4,5,6};
Print((int*)s,6);//(int*)s==s[0];
return 0;
}
void Print(int **p)
```

```
{
    for (int i = 0; i < 2; i++)
    {
        for (int j = 0; j < 3; j++)
            printf("%d ", *(( int* )p+ 2*i + j));//编译器寻址方式，根据二级指针不能自动寻址
    }
    printf("\n");
}

int main()
{
    int s[2][3] = {1, 2, 3, 4, 5, 6};
    Print((int **)s);
    return 0;
}
```

二维数组名的类型是type (*)[n];是一个数组指针，可强制转换成type**和type*，地址是一样的

常用

int s[][n];n不能为空。一维数组也建议用int s[];较直观，便于与其他的指针区分。

动态内存分配

```
 int **s = (int **)malloc(2 * sizeof(int *));
    for (int i = 0; i < 2; i++)
        s[i] = (int *)malloc(3 * sizeof(int));
        for(int j = 0; j <2;j++)
        {
            for(int k = 0;k<3;k++)
            s[j][k]=1;
        }
for(int j = 0; j <2;j++)//逐层由内向外释放
        free(s[j]);
        free(s);

 int *s[2];
    for (int i = 0; i < 2; i++)
        s[i] = (int *)malloc(3 * sizeof(int));
        for(int j = 0; j <2;j++)
        {
            for(int k = 0;k<3;k++)
            s[j][k]=1;
        }
        for(int j = 0; j <2;j++)//不用也不能释放s，它不是动态分配的
        free(s[j]);
```

A&&B和A||B的短路规则

如果A为假，则A&&B短路，如果A为真，则A||B短路，不再判断B。

应用：对于A||B如果A真时B可能溢出则A必须在前；

查看函数中数组的值

1，*(type(*)[len])arrname;强制转换为数组指针（它的最本质类型），再取值。

2,*(arrname)@len;用这个简单。

删除字符

```
void delchar( char *str, char c )
{
    char *p=str;int i=0,j=0;//p和str指向同一条字符串，但p在str后面进行判断所以不受干扰
    while(p[i])
    {
        if(p[j]!=c)
        str[i++]=p[j];
        j++;
    }
}
```

进制转换

```
void ten2two(int n)
{
if(n>0)
{
ten2two(n/2);
printf("%d",n%2);        //递归为逆序输出，故先写调用函数，后写printf
}
else
return;
```

```
int convert(int n)
{
    if(n==0||n==1)
        return n;
    else
        return n%10+2*convert(n/10);
}
```

选择排序法小改进

```
 for(int j=0,index;j<i-1;j++)
    {
        index=j;
        for(int k=j+1;k<i-1;k++)
        {
            if(strlen(s[k])<strlen(s[index]))
                index=k;
        }
        printf("%s ",s[index]);//如果只是要输出将最小的依次输出然后把最小值替换即可，不用交换
        strcpy(s[index],s[j]);
    }
```

获取随机数

从X到Y，有Y－X＋1个数，所以要产生从X到Y的数，只需要这样写： *k=rand()%(Y-X+1)+X;*

每次获取前运行*srand(time(NULL))*；改变随机数计算函数初值，在time.h下。

y=x++和y=++x都是自增运算符。区别在于：y=x++是先把x的值赋给y，此时y的值为5。赋值给y后，x自增，即x=x+1，此时x的值就变成6了。y=++x是x自增，即x=x+1，此时x的值为6。自增完后，x的值赋给y，此时y的值就变成6了。综上来看，也就是y=x++相当于，y=x，x=x+1。y=++x相当于，x=x+1，y=x。（执行的先后顺序不一样）

一字节（byte）等于八位（bit）；

整数取值范围：\overbrace{100000....000}^{31}=-2^{31} 补码是源码**除符号位**外取反加一，逆推原码也是这样或者减一取反。\overbrace{0111111111111111}^{32} =2^31-1，具体值用**pow表达式或者用确切值**。

用C++写的时候可以用new和delete以及引用传参

算法和算法设计

算法：一个有穷的指令集。

算法的特性：

1，有输入：可以通过输入语句**由外部显式提供**，也可以**由赋值或定值语句隐式提供**，即 **“0个输入“** 情况； 2，有输出； 3，确定性：每一步都确切，无歧义的定义，**对于一组确定的输入对应一条确定的路径运算**，如果会因系统状态而导致结果不一致，则只要对于每个系统状态有确定的处理手段就不影响确定性，没赋初值导致的结果不同没有对应不同状态的处理，违反确定性。 4，有穷性：**初值导致算法不收敛的与算法本身无关，不违反有穷性**； 5，可行性：每一条运算都足够基本（**可以用基本操作或调用已实现的基本算法**），都能精确执行，但**并不一定都与机器指令有直接关系**，并**能在常数时间内完成**。

基本设计步骤：**理解需求->设计思路->算法框架->程序实现**；

算法与程序的关系： (1)算法在描述上一般使用半形式化的语言，而程序是用形式化的计算机语言描述的。 (2) 程序是计算机指令的有序集合。 (3)程序并不都满足算法所要求的特征，例如操作系统，是一个在无限循环中执行的程序，因而不是一个算法。 (4)一个算法可以用不同的编程语言编写出不同的程序。 (5)算法是解决问题的步骤；程序是算法的代码实现。 (6)算法要依靠程序来完成功能；程序需要算法作为灵魂。 (7)程序=算法+数据结构。

算法设计基本方法：

1，穷举法（枚举法）：<1>按规则列举，<2>盲目列举，并检查之前的列举是否重复；

2，迭代法（反复法）:iteration：不断用原值得到的新值代替原值，直到得到满意的解，新值与原值之间的关系用迭代公式表示，主要用于很难用或无法用解析法求解的计算问题，例如区间折半法求方程的根，也用来遍历表树图等数据结构。

3，递推法：递归的递推求解使用递归法，自顶向下，非递归的递推求解使用迭代法，自底向上；

4，递归法：包括自身的数据对象和调用本身的过程是递归的；递归从问题规模为n的场合开始，通过递归降低问题规模，直到递归出口，再倒推回来得到最初的值；递推是从已知条件出发；一般一个递推算法总可以转化为递归算法；例如二分法求根和求Fibonacci数两种方式的转换。但递归法不仅仅用于递推的实现（还有数据结构如链表，树图等的建立等等）。

递归是算法设计的基本技术，是降低分析设计难度提高程序设计效率的重要手段和工具；迭代具有更高的时空效率。

穷举过程中被穷举对象可能需要其他方法求解，各种方法是配合使用的。

算法度量

算法的评价标准： 1，正确性； 2，健壮性：在不正确输入条件下能自我保护，包括自动检错，报错，与用户对话来纠错； 3，可读性；变量名，函数名要有实际意义，必须加入注释； 4，高效性：主要指算法的时间代价和空间代价； 5，简单性：主要用环路复杂度度量，等于程序中判断语句和子程序调用总数加一，软件工程要求不能超过10；

算法分析的主要方法：**事后*估算法***（插入*测量时间语句*来估算）和**事前*统计法***（通过对*问题规模，执行频度，时间，空间复杂度*的进行估算。**主要目的是分析算法的效率以求改进。主要方面是时间性能和空间性能**。

测量程序运行时间

clock_t start_time, end_time;给两个变量赋给当时时间，它们的差就是它们之间程序的运行时间

clock_t是长整形。

用clock()(精确到豪秒）或者time(NULL/0)（精确到秒）

用time直接作差即可，用clock的需要使用表达式(double)(end_time-start_time)/CLOCK_PER_SEC(每秒钟clock的增量，linux下为1000000，window下为1000)；

特别的sleep(x)（unisted.h)下会使linux下的clock暂停(它返回的是CPU耗费在本程序上的时间。也就是说，途中sleep的话，由于CPU资源被释放，那段时间将不被计算在内。)等待输入时clock也会暂停，带有sleep的程序运行时间精度要求不高时只能用time；windows不会；

window下sleep(n)单位是ms，linux下单位是s；linux下还有usleep(n),单位是um。里面乘以1000就等同于window下的sleep。

精确计算运行时间

```
#include <sys/time.h>
struct timeval  start_time,end_time;
{
 gettimeofday(&start_time,NULL);
 /*
 代码块
      */
gettimeofday(&end_time,NULL);
printf("%lf",(end_time.tv_sec-start_time.tv_sec)+(double)(end_time.tv_usec-
start_time.tv_usec)/CLOCKS_PER_SEC);

    return 0;

}
/*
gettimeofday()会把目前的时间用tv 结构体返回，当地时区的信息则放到tz所指的结构中。
一般情况下 ，我们并不需要时区信息，所以第二个参数通常为空。
timeStart.tv_sec 这个就是秒为单位的时间戳。(double)
timeStart.tv_usec 这是当前秒中的毫秒数。(int,需要除以CLOCK_PER_SEC并转为double)；
*/
```

算法的计算量的大小称为算法的*复杂性*。

算法的复杂度(complexity)whe.,,S度量和问题规模

问题规模从问题的描述中找到：例如在n个学生中查找和求解n阶线性方程组的问题规模都是n；

时间复杂度T(n)：当问题的规模从1增加到n时，解决问题的算法所耗费的时间也由1增加到T(n);

空间复杂度S(n):空间由1到S(n);

两种度量都是问题规模n的函数，单位都是1，即单位时间(ont time unit)和单位空间都是1(ont space unit);

时间复杂度度量的计算

算法的执行频度=每条语句的执行次数（频度）x该语句执行时间（每一条基本语句执行时间视为单位时间1，语句执行时间等于语句中基本运算语句数）=算法中所有**运算语句**执行频度的总和;

for循环控制语句的执行次数为n+1,；单位执行时间（一次执行所需语句数）为2（不包括内部），执行频度（总次数）为2(n+1)(共执行n+1次表达式2，1次表达式1，n次表达式3），循环体执行次数是n;

运算赋值语句是一个基本运算语句（**加减乘除，转移，存取以及他们的复合**），如a=b+c，return b+c，执行频度都是1；定义语句不是运算语句;

递归算法的执行频度可通过写出T(n)的递推形式来计算；例如T(n)=2,n<=0（if判断和return）;T(n)=2+T(n-1);n>0;（比上次多两次执行（if和return加和，**调用函数以及return的执行次数和加和合并了，还是1**）；

空间复杂度度量指算法所需附加存储空间，包括固定部分和可变部分（与问题规模有关）；

阶乘的非递归实现中，只用了一个整数存放连乘结果，附加空间数为1，空间复杂度也为1;

递归的空间复杂度=每次递归所要的辅助空间x递归深度；

阶乘的递归i实现中每一层递归都需要三个栈空间来存放形式参数，返回值以及返回地址（一旦函数调用完成，程序应该知道返回的位置，即函数调用之前的点）

递归深度是n，所以所需的栈空间是3n，空间复杂度为3n；

动态分配所涉及的空间复杂度 等于malloc的空间减去free的空间；

渐进的时间复杂度

执行频度不能确切地反映运行时间，所以用其来比较两个程序结果不一定有价值，所以只需给出算法执行频度的数量级即可达到分析的目的。

大*O*表示：当且仅当存在正整数c和n_0,使得T(n)<=cxf(n)对所有的n>=n_0成立，则称该算法的时间增长率在*O*(f(n))中，记为T(n)=*O*(f(n)).

算法时间复杂度T(n)增长的上限为f(n);*Never to underestimate the runing time of the program.*

O 函数的渐近上界 upper bound Ω 函数的渐近下界 lower dound Θ 函数的准确界

Θ(f(n))=T(n):存在正常数c1,c2和n0，使对所有的n⩾n0，有0⩽c1f(n)⩽T(n)⩽c2f(n)
f(n)=2n^2+n,T(n)=n^2;(同阶)同速

O(f(n))=T(n): 存在正常数c和n0，使对所有n⩾n0，有0⩽T(n)⩽cf(n) f(n)=n^2;T(n)=2n^2+n;
f(n)_rate>=T(n)_rate

Ω(f(n))=T(n): 存在正常数c和n0，使对所有n ⩾ n0，有0⩽cf(n)⩽T(n)
f(n)=n^2,T(n)=2n^2+n;f(n)_rate<=T(n)_rate

o(f(n))=T(n): 对任意正常数c，存在常数n0>0，使对所有的n⩾n0，有0⩽T(n)⩽cf(n)
T(n)=O(f(n))&&T(n)!=Ω(f(n));f(n)=2n^2+n;T(n)=2n;

f(n)_rate>T(n)_rate

lim(T(n)/f(n))=*lim(T'(n)/f'(n))*(n->inf):1,==c:Θ(f(n))=T(n);2,->inf:o(T(n))=f(n);3,==0;o(f(n))=T(n);

Thus T(n)=n^2=O(n^2)=O(n^3),the first option is the best answer.

一般情况下O就是指Θ，前者范围更广所以一般用前者表示。

If T(N)is a polynomial(/ˌpɑːliˈ noʊmiəl/ 多项式) of degree k(equal to k-order),then T(N)= Θ(N^k);

Lower-order terms can generally be ignored,and the **constants** can be throw away.

If T1(n) = O(F(N)) and T2(N) = O(g(N)),then (a),T1(N)+T2(N) = max{O(f(N)),O(g(N))}
(b),T1(N)*T2(N) = O(f(N)*g(N)).

When n is sufficiently large, the growth of various functions has the following relationship:

c < log2n < (log2n)^k<n < nlog n < n^2 < n^3 < 2^n < 3^n < n! <n^n

Constant logarithmic Linear Quadratic Cubic Exponential factorial /ˌekspəˈ nenʃl/ The logarithms grow very slowly.

$$\log_{k_1} n < n^{k_2} < {k_3}^n (k_1, k_2 > 0, k_3 > 1)$$

the rate of **logarithm slower than power slower than exponient,there is no power function between n and nlogn.**

such as :

$$N^{1+\frac{\varepsilon}{\sqrt{\log n}}} < N \log N$$

O(log2n)可以简记为O(logn);由换底公式知不同底数的对数阶只差了常数倍，n(log2n)也满足

for loop statement:The total running time of a statement（语句） inside a group of nested loops(嵌套循环) is the running time of statement multiplied by the product（乘积） of the size of all the for loops;

if/else statement:time of *test* plus the *lager* of the running time of S1 and S2.

函数或语句嵌套的相乘，并列的取最大。

计算递归的时间复杂度时，简单的可以直接当作for循环来看，复杂的通过递推式计算，结束递归那一步的执行频度如果是c可以简化为1，递推式中的常数必须严格按其执行频度来，例如裴波那契数列递归式是T(n)=T(n-1)+T(n-2)+2(if和return那一句)，n=1,2时看作1还是2无所谓。

数学公式

在线生成 一个行内，两个行外； *下标，* ^*上标 逻辑上的括号:{} 分数：* \frac{}{} *方程组:* *start{cases}* \ \ \ *end{cases}* *求和:*\sum{i=0}^k; 换行\,空格; 箭头\rightarrow 自适应括号\left(\right) \sqrt{} \overbrace{a+\underbrace{b+c}_{1.0}+d}^{2.0}

名称	缩写
JavaScript	js
Python	py
C++	cpp

“: ”: 决定对齐方式

题号	标题	难易度
1	两数之和	简单
15	三数之和	中等
262	行程和用户	困难

时间复杂度例题：

循环嵌套

```
{
int  i,j,k,x=0,y=0;
for(int i=1;i<=n;i++)
for(int j=1;j<=i;j++)
for(int k=1;k<=j;k++)
x=x+y;
}
```

T(n) = \sum_{i=1}^n \sum_{j=1}^i \sum_{k=1}^j = \sum_{i=1}^n \sum_{j=1}^i j = \sum_{i=1}^n \frac{i(i+1)}{2} = \frac{1}{2}\times\frac{n(n+1)(2n+1)}{6}+\frac{1}{2}\times\frac{n(n+1)}{2} = \frac{n(n+1)(n+2)}{6}\\ \sum_{i=1}^n i^2=\frac{n(n+1)(2n+1)}{6}\\ \sum_{i=1}^n i^3=\left(\sum_{i=1}^n i\right)^2;

```
{
    int func(int n)
    {
    int i=1,s=1;
    while(s<n)
    s+=++i;
    return i
    }
}
第一轮 ：s=1+2;第二轮：s=1+2+3；第n轮：s=1+2+3+...+i;
```

跳出循环时，函数执行次数i满足 \frac{n(n- 1)}{2}\geq n;即 i^2-i-2n\geq0,解得 i\geq\frac{1\pm\sqrt{1+8n}}{2}.故时间复杂度为O(\sqrt{n});

Find the maximum of subsequence(substring) sum in A[n]:{A0,A1,...An};

$$\sum_{k=i}^{j} A_k$$

```
Algorithm 1
int MaxSubsequenceSum(const int A[],int N)
{
    int ThisSum,Maxsum,i,j,k;
    Maxsum=0;
    for(i=0;i<N;i++)
    for(j=i;j<N;j++)
    {
        ThisSum=0;
        for(k=i;k<=j;k++)
        ThisSum+=A[k];
        if(ThisSum>Maxsum)
        Maxsum=ThisSum;
    }
```

```
    return Maxsum;
}
```

$$\sum_{i=0}^{N}\sum_{j=i}^{N}\sum_{k=i}^{j}=\frac{N^3+3N^2+2N}{6}$$

evaluated inside out（由内到外计算）

求内层变量和时外层的变量当常量对待。

时间复杂度为O(n^3);不需要精确计算时可以通过三层循环的次数都小于等于n得到结果；

```
Algorithm2:
int MaxSubsequenceSum(const int A[],int N)
{
    int ThisSum,Maxsum,i,j,k;
    Maxsum=0;
    for(i=0;i<N;i++)
    {
        ThisSum=0;
    for(j=i;j<N;j++)
    {
       ThisSum+=A[j] ;
        if(ThisSum>Maxsum)
        Maxsum=ThisSum;
    }
    }
    return Maxsum;
}
clearly is O(n^2) ;
```

```
Algorithm3:

#include <iostream>
int Max3(int a, int b, int c);
int MaxSubSum(const int A[], int Left, int Right);
int main()
{
    int a[5] = {1, 3, -5, 0, -8};
    printf("%d", MaxSubSum(a, 0, 4));
    return 0;
}
int Max3(int a, int b, int c)
{
    if (a > b)
        return a;
    else if (c > b)
        return c;
    else
        return b;
}

int MaxSubSum(const int A[], int Left, int Ri ght)
{
    int MaxLeftSum, MaxRightSum;
    int MaxLeftBorderSum, MaxRightBorderSum;
```

```
    int LeftBorderSum, RightBorderSum;
    int Center, i;
    if (Left == Right)//递归返回终点
        if (A[Left] > 0)
            return A[Left];
        else
            return 0;
    Center = (Left + Right) / 2;
    MaxLeftSum = MaxSubSum(A, Left, Center);//递归计算两边的最大子列和，分治思想体现。
    MaxRightSum = MaxSubSum(A, Center + 1, Right);
    MaxLeftBorderSum = 0;
    LeftBorderSum = 0;
    for (i = Center; i >= Left; i--)
    {
        LeftBorderSum += A[i];
        if (LeftBorderSum > MaxLeftBorderSum)
            MaxLeftBorderSum = LeftBorderSum;
    }
    MaxRightBorderSum = 0;
    RightBorderSum = 0;
    for (i = Center + 1; i <= Right; i++)
    {
        RightBorderSum += A[i];
        if (RightBorderSum > MaxRightBorderSum)
            MaxRightBorderSum = RightBorderSum;
    }
    return Max3(MaxLeftSum, MaxRightSum, MaxLeftBorderSum + MaxRightBorderSum);
                // 最大值在这三个值里产生
}
```

The algorithm uses a "divide-and-conqure"(分治) strategy.The idea is spit the problem into two roughly equal subproblems,which are the solved recursively.

An Algorithm is**O(log(N))if it takes constant(O(1))time to cut the problem size by a *fraction*(which is usually 1/2).On the other hand,if constant time is required to merely reduce the problem by a *constant amount*(such as make the problem smaller by 1),then the algorithm isO(N)**.

$$\begin{cases} T(n) = T(n/2) + O(1) \rightarrow T(n) = kO(1) = O(1)\log n = O(\log n) \\ T(n) = 2(T/2) + O(1) \rightarrow T(n) = 2^k + kO(1) = n + O(1)\log n = O(n) \\ T(n) = 2(T/2) + O(n) \rightarrow T(n) = 2^k + n \times k = n + n\log n = O(n\log n) \end{cases}$$

Simple intuition obviates the need for a brute-force approach.

接收n个数据至少需要O(N)的复杂度，故O(logN)是针对对应函数而不是程序整体说的。

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。

分治策略是：对于一个规模为n的问题，若该问题可以容易地解决（比如说规模n较小）则直接解决，否则将其分解为k个规模较小的子问题，这些子问题互相独立且与原问题形式相同，递归地解这些子问题，然后将各子问题的解合并得到原问题的解这种算法设计策略叫做分治法。

如果原问题可分割成k个子问题，1<k≤n，且这些子问题都可解并可利用这些子问题的解求出原问题的解，那么这种分治法就是可行的。由分治法产生的子问题往往是原问题的较小模式，这就为使用递归技术提供了方便。在这种情况下，反复应用分治手段，可以使子问题与原问题类型一致而其规模却不断缩小，最终使子问题缩小到很容易直接求出其解。这自然导致递归过程的产生。分治与递归像一对孪生兄弟，经常同时应用在算法设计之中，并由此产生许多高效算法。

$$\begin{cases} T(n)=1, n=1 \\ T(n)=2 * T(n/2)+O(n), n>1 \ if \ n=2^k, then \ T(n)=2^k+n\times k=n+n\log_2 n=O(Nlogn) \end{cases}$$

```
Algorithm4
int MaxSubsequenceSum(const int A[],int N)
{
    int ThisSum,Maxsum,j;
    ThisSum=MaxSum=0;Maxsum=0;
    for(j=0;j<N;j++){
        ThisSum+=A[j];
        if(ThisSum>MaxSum)
        MaxSum= ThisSum;
        else if(ThisSum<0)//越过分界点时ThisSum归零，计算下一段的最长字列和
        ThisSum=0;
    }
    return Maxsum;
}
```

首先假设我们已经找到了最大连续和子串在数组中的起始位置（i）和结束位置（j），其中i <= j，即最大和maxSum = a[i] + a[i + 1] + ... + a[j]，我们来看看这个子串有什么性质： a[i] > 0，否则我们完全可以去掉a[i]这个元素 而得到一个更大的和； i > 0且a[i - 1] < 0 或 i == 0，下面假设i > 0，这一条性质也是因为如果a[i - 1] > 0的话我们求和时可以加上a[i - 1]这个元素得到一个更大的和；元素a[i - 1]与它之前的任一元素之间的子串之和sum < 0，即对于任何一个m(0 <= m < i - 1)，则有a[m] + a[m+ 1] + ... + a[i - 1] < 0，**（最大序列和不可能越过分界点，并且前面的最大字列的终点一定是m-1（如果存在非零子列和的话）一定会被记在MaxNum里）**，这条性质同样可以用反证法证明。

只要找到分界点 a[i - 1]，最大和的子串要么就在a[i - 1]元素之后，要么就在a[i - 1]之前，最大和的子串不可能跨过a[i - 1]这个点；一个数组中可能有多个这种分界点，但每个分界点都可以把前后完全分开，可以单独算分界点之间的最大和，然后在这些最大和之间取最大值。 假设对于数组a，我们找到了两个分界点a[i]和a[j],那么整个数组的最大字串和就是max(sum(a[0]...a[i-1]), sum(a[i+1]...a[j-1]), sum(a[j+1]...a[len-1]))。

Binary Search(二分查找)

```
//A[N] has presorted(一次排序，永远方便，需要多次查找的都可以这样搞)
int BinarySearch(Const ElementType A[],ElementType X,int N)
{
    int Low,Mid,High;
    Low=0;High=N-1;
    while(Low<=High)
```

```
    {
        Mid=(Low+High)/ 2;
        if(X>A[Mid])
        Low=Mid+1;
        else
        if(X<A[Mid])
        High=Mid-1;
        else
        return Mid;//Found
    }
    return NotFound//is defined as -1;
}
```

每次循环使问题的规模减小一倍，故时间复杂度是O(logn);

最大公约数和最小公倍数以及 二分法求根

```
int GCD(int a , int b )//辗转相除法(Greatest Common Divisor)
{
    int temp;
    if(!(a%b))
    { return b;break;}
    else
        return GCD(b,a%b);//除数除余数
}
//都不用判断a，b大小，如果a<b,第一次递归或迭代会交换他们。
```

迭代写法（推荐）：

```
int GCD(int M, int N) //(Euclid's algorithm)
{
    int Rem;
    while(N>0)
    {
        Rem=M%N;
        M=N;
        N=Rem;
    }
    return M;
}
```

if M>N,then M mod N < M/2;(If N<=M/2,then remainder is smaller than N;if N>M/2,then remainder is M-Nalso smaller than M/2);

考虑(a,b)⟶(b,a mod b)这个迭代，有两种情况： 1，如果a>b,那么迭代两次后得到(a mod b,b mod (a mod b)), 有a mod b < $\frac{a}{2}$, b mod (a mod b) < $\frac{b}{2}$, 即迭代两次后问题的规模减小一倍； 2，如果a>b,迭代一次后归入情况a，至多出现一次，可以忽略。 故其时间复杂度为 $2\log n = O(\log n)$;

```
}
int gcd(int m,int n)更相减损术
{
    if(m>n)
    return gcd(m-n,n);
    else if(n>m)
    return gcd(m,n-m);//大数减小数
```

```
    else
    return m;
}
```

设gcd(a,b)=c,lcm(a,b)=d，即ab的最大公因数为c，最小公倍数为d，则一定有a=k1*c* ， *b=k2*c （1） a=d/t1 b=d/t2 （2）,其中k1,k2必然互质（反证法：假设k1，k2不互质，则必定有一个大于1的最大公因数，设其为x，则有a=k1/x * xc, b=k2/x * xc,此时k1/x和k2/x已经互质，但是明显a和b有一个公因数xc,又因为x>1，则xc>c，和前提ab的最大公因数c矛盾，则k1,k2必然互质，同理可证t1,t2互质）

对于（1）式，可有b/a=k2/k1.对于（2）式，可有b/a=t1/t2,即k2/k1=t1/t2,定有常数y,使得k2=y*t1,k1=y*t2,即k2/k1=y*t1/y*t2=t1/t2,

而k2和k1是互质的，y只能为1，所以k2=t1,k1=t2,而k1=a/c,t2=d/b,所以a/c=d/b,即ab=cd，证毕

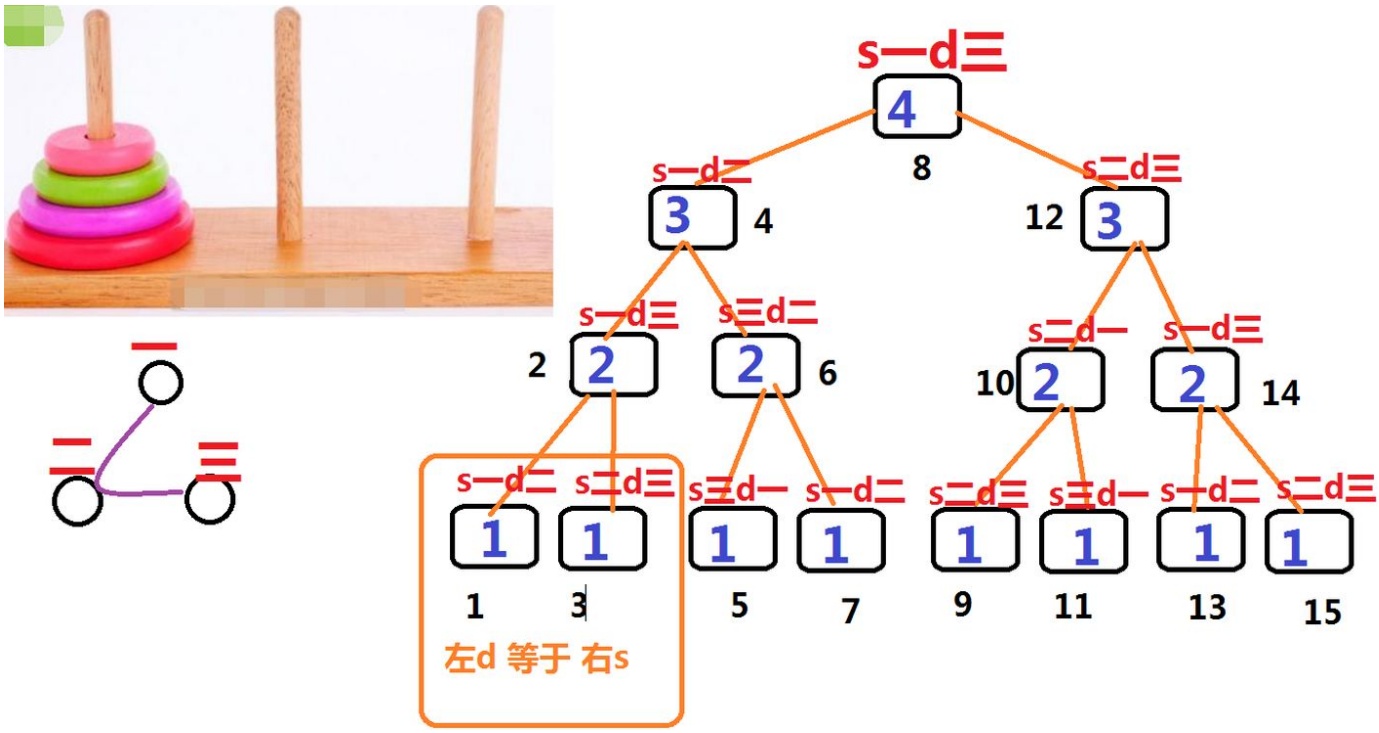
```
double f(double x)
{
    return a3*(x)*(x)*(x)+a2*(x)*(x)+a1*(x)+a0;
}
double isroot(double a,double  b)
{
    if(fabs(f((a+b)/2))<1e-6)//也可用区间长度作为控制精度结束条件
        return (a+b)/2;
    if(f((a+b)/2)*f(a)<0)
        return isroot(a,(a+b)/2);
    else
        return isroot((a+b)/2,b);

}
```

```
//Exponentiation:
long int Pow(long int x,int N)
{
    if(N==0)
    return 1;
    if(N==1)//unneceaaary
    return 1;
    if(IsEven(N))
    retuen Pow(x*x,N/2);
    else
    retuen Pow(x*x,N/2)*x;
}
O(log N)
```

汉诺塔

```
void hanoi(int n, char A, char B, char C)//盘子数量，初始轴，中间轴，目标轴；
{
    if (n == 1)//!最频繁的操作，是两个递归调用的终止条件
```

```
    {
        move(1,A, C);
        return;//或else
    }
    hanoi(n - 1, A, C, B);//将A上面n-1个移动到B
    move(n,A, C);//将A上最后一个最大的移动到C，也是移动每个子递归上最大的；
    hanoi(n - 1, B, A, C);//将中间轴B上的移动到C
}
void move(int n,char A, char C)
{
    cout << m++<<':'<<n<<"from" << A << " to" << C << endl;
}
```



先从左边三号节点进入，到最左边的一号节点开始逐层退出递归，全部退出后执行节点4，然后从右边三号节点进入到其左下角的一号节点开始逐层退出递归。

$$T(n) = 2T(n-1) + k \rightarrow T(n) = 2^n$$

随机置换数组

```
Algorithm1:
int A[MAXN] = {0},i,j,temp;
    for (i = 0; i < MAXN; )
    {
        srand(time(NULL));
        temp = rand() % 10 + 1;
        for (j = 0; j < i; j++)
        {
            if (temp == A[j])
                break;
        }
        if(j==i)
        A[i++]=temp;
    }
```

The expected number of random numbers that need to be tried is N/(N-i), This is obtained as follows: i of the N numbers would be duplicates.Thus the probability of success is N-i/N(if成功的概率); Thus the expected number of independent trials is N/N-i（if 成功一次所需的次数）;在x次独立重复事件中，该事件发生xp次；所以该事件发生一次的概率是1/p（两边同除以xp) The time bound is thus:

$$\sum_{i=0}^{i-1} \frac{Ni}{N-i} < \sum_{i=0}^{N-1} \frac{N^2}{N-i} = N^2 \sum_{i=0}^{N-1} \frac{1}{N-i} = \sum_{j=1}^{N} \frac{1}{j} = N^2 \log N$$

The time bound is thus

```
Algorithm2:

int A[MAXN] = {0},Used[MAXN]={0},i,j,temp;
    for (i = 0; i < MAXN; )
    {
        srand(time(NULL));
       temp=rand()%MAXN+1;
       if(!Used[temp-1])
       {
           A[i++]=temp;Used[temp-1]=1;
       }

    }
```

Obviously time complexity is O(nlogn);

KMP

P 的 next 数组定义为：next[i] 表示 P[0] ~ P[i-1] 这一个子串，使得 前k个字符恰等于后k个字符 的最大的k. 特别地，k不能取i+1（因为这个子串一共才 i+1 个字符，自己肯定与自己相等，就没有意义了）。

性质：P[0] 到 P[i] 这一段子串中，前next[i]个字符与后next[i]个字符一模一样。既然如此，如果失配在 P[r], 那么P[0]~P[r-1]这一段里面，前next[r]个字符恰好和后next[r]个字符相等——也就是说，我们可以拿长度为 next[r] 的那一段前缀，来顶替当前后缀的位置，让匹配继续下去.

主串影响失配的位置和多少，模式串影响回溯的步数（普通算法就是回溯到串首）

求next数组的过程完全可以看成字符串匹配的过程，即以模式字符串为主字符串，以模式字符串的前缀为目标字符串，一旦字符串匹配成功，那么当前的next值就是匹配成功的字符串的长度。具体来说，就是从模式字符串的第一位(注意，不包括第0位)开始对自身进行匹配运算。 在任一位置，能匹配的最长长度就是当前位置的next值。最大步增为1，最小值为0；

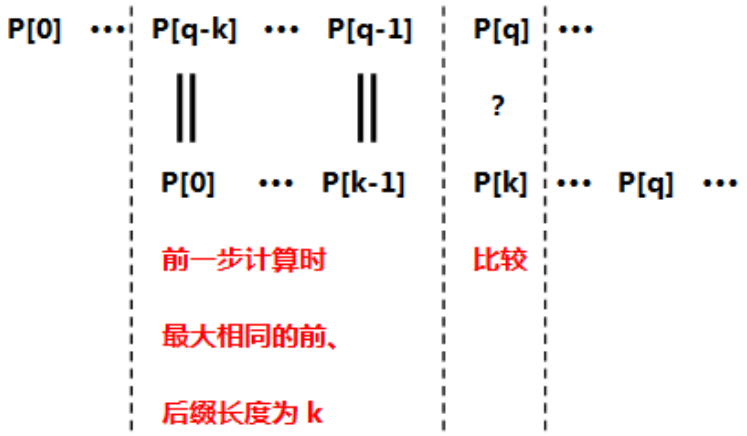


图 1

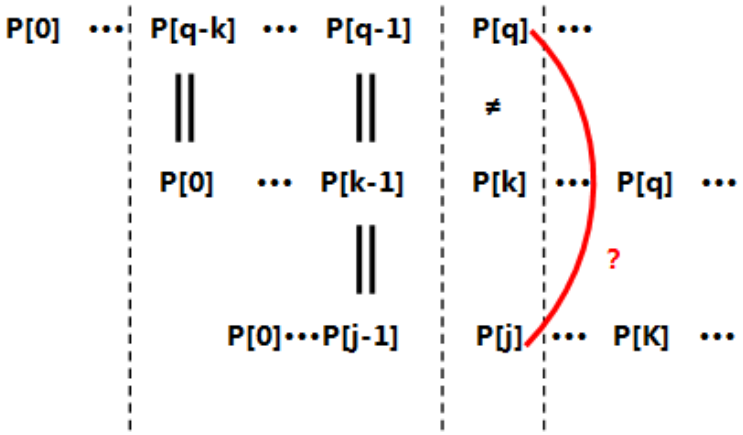


图 2

```
void getNext(char * p, int * next)
{
        next[0] = -1;
        int i = 0, j = -1;

        while (i < strlen(p))
        {
                if (j == -1 || p[i] == p[j])
                {
                        ++i;
                        ++j;
                        next[i] = j;
                }
                else
                        j = next[j];
        }
}

int KMP(char * t, char * p)
{
        int i = 0;
        int j = 0;

        while (i < strlen(t) && j < strlen(p))
        {
                if (j == -1 || t[i] == p[j]) //除了初始状态，当p[0]与失配的t[i]匹配不上时j会等于-1，然后p[0]将于失配处的下一个字符进行匹配，特别的i从来不会回退，也不需要回退，next数组的本质就是利用已经配对过的部分控制模式串的偏移量
                {
                        i++;
```

```
                        j++;
                }
                else
                        j = next[j];
        }

    if (j == strlen(p))
       return i - j;模式串的位置
    else
       return -1;
}
```

广义表

广义表，又称列表，也是一种线性存储结构。同数组类似，广义表中既可以存储不可再分的元素，也可以存储广义表，记作：

LS = (a1,a2,…,an)

其中，LS 代表广义表的名称，an 表示广义表存储的数据。广义表中每个 ai 既可以代表单个元素，也可以代表另一个广义表。

通常，广义表中存储的单个元素称为 "原子"，而存储的广义表称为 "子表"。

例如创建一个广义表 LS = {1,{1,2,3}}，我们可以这样解释此广义表的构成：广义表 LS 存储了一个原子 1 和子表 {1,2,3}。

以下是广义表存储数据的一些常用形式： A = ()：A 表示一个广义表，只不过表是空的。 B = (e)：广义表 B 中只有一个原子 e。 C = (a,(b,c,d))：广义表 C 中有两个元素，原子 a 和子表 (b,c,d)。 D = (A,B,C)：广义表 D 中存有 3 个子表，分别是A、B和C。这种表示方式等同于 D = ((),(e),(b,c,d)) 。 E = (a,E)：广义表 E 中有两个元素，原子 a 和它本身。这是一个递归广义表，等同于：E = (a,(a,(a,…)))。

注意，A = () 和 A = (()) 是不一样的。前者是空表，而后者是包含一个子表的广义表，只不过这个子表是空表。

当广义表不是空表时，称**第一个数据（原子或子表）为"表头"，剩下的数据构成的新广义表为"表尾"**。

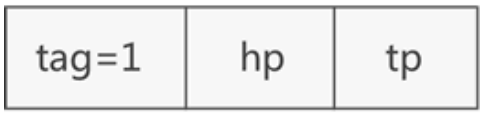
强调一下，除非广义表为空表，否则广义表一定具有表头和表尾，且广义表的表尾一定是一个广义表。

例如在广义表中 LS={1,{1,2,3},5} 中，表头为原子 1，表尾为子表 {1,2,3} 和原子 5 构成的广义表，即 {{1,2,3},5}。

再比如，在广义表 LS = {1} 中，表头为原子 1 ，但由于广义表中无表尾元素，因此该表的表尾是一个空表，用 {} 表示。

tag 标记位用于区分此节点是原子还是子表，通常**原子的 tag 值为 0，子表的 tag 值为 1**。子表节点中的 **hp 指针用于连接本子表中存储的原子或子表，tp 指针用于连接广义表中下一个原子或子表**。

A) 原子结点



B) 表结点

```
typedef struct GLNode{
    int tag;//标志域
    union{
        int atom;//原子结点的值域
        struct GLNode *hp;//子表结点的指针域，hp指向表头
    };//共用体不用命名，直接引用其中成员即可
    struct GLNode * tp;//这里的tp相当于链表的next指针，用于指向下一个数据元素
}*Glist;
```

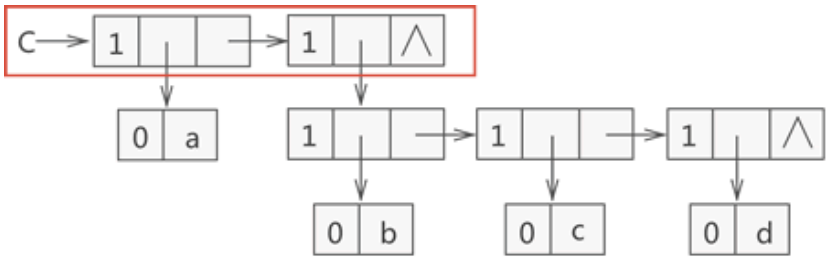
```
Glist creatGlist(void){
   Glist C = new GLNode;
    C->tag=1;
    C->hp=new GLNode;
    C->tp=NULL;
    //表头原子a
    C->hp->tag=0;
    C->hp->atom='a';
    C->hp->tp=new GLNode;
    C->hp->tp->tag=1;
    C->hp->tp->hp=new GLNode;
    C->hp->tp->tp=NULL;
    //原子b
    C->hp->tp->hp->tag=0;
    C->hp->tp->hp->atom='b';
    C->hp->tp->hp->tp=new GLNode;
    //原子c
    C->hp->tp->hp->tp->tag=0;
    C->hp->tp->hp->tp->atom='c';
    C->hp->tp->hp->tp->tp=new GLNode;
    //原子d
    C->hp->tp->hp->tp->tp->tag=0;
    C->hp->tp->hp->tp->tp->atom='d';
    C->hp->tp->hp->tp->tp->tp=NULL;
    return C;
}
```

由于广义表中可以同时存储原子和子表两种类型的数据，因此在计算广义表的长度时规定，**广义表中存储的每个原子算作一个数据，同样每个子表也只算作是一个数据**。

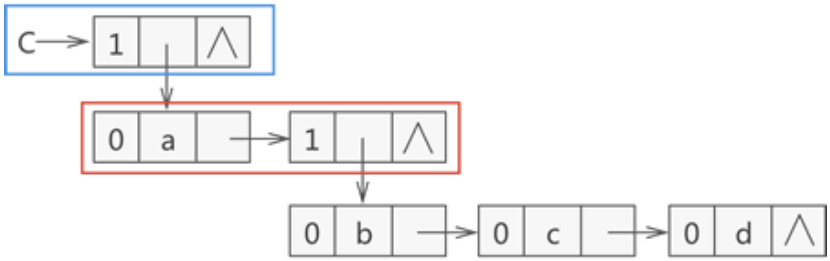
例如，在广义表 {a,{b,c,d}} 中，它包含一个原子和一个子表，因此该广义表的长度为 2。

再比如，广义表 {{a,b,c}} 中只有一个子表 {a,b,c}，因此它的长度为 1。

前面我们用 LS={a1,a2,...,an} 来表示一个广义表，其中每个 ai 都可用来表示一个原子或子表，其实它还可以表示广义表 LS 的长度为 n。 广义表规定，空表 {} 的长度为 0。



a) 广义表存储结构示意图



b) 另一种广义表存储结构示意图

第一种将原子也看作字表，通过字表互相连接；第二种将两者区分开，同级的互相连接

对于图 1a) 来说，只需计算最顶层（红色标注）含有的节点数量，即可求的广义表的长度。**C指向表头**

同理，对于图 1b) 来说，由于其最顶层（蓝色标注）表示的此广义表，而第二层（红色标注）表示的才是该广义表中包含的数据元素，因此可以通过计算第二层中包含的节点数量，即可求得广义表的长度。**C指向整体这个表**;

```
int GlistLength(Glist C){
    int Number=0;
    Glist P=C->hp;//第一种这里是C
    while(P){
        Number++;
        P=P->tp;
    }
    return Number;
}
```