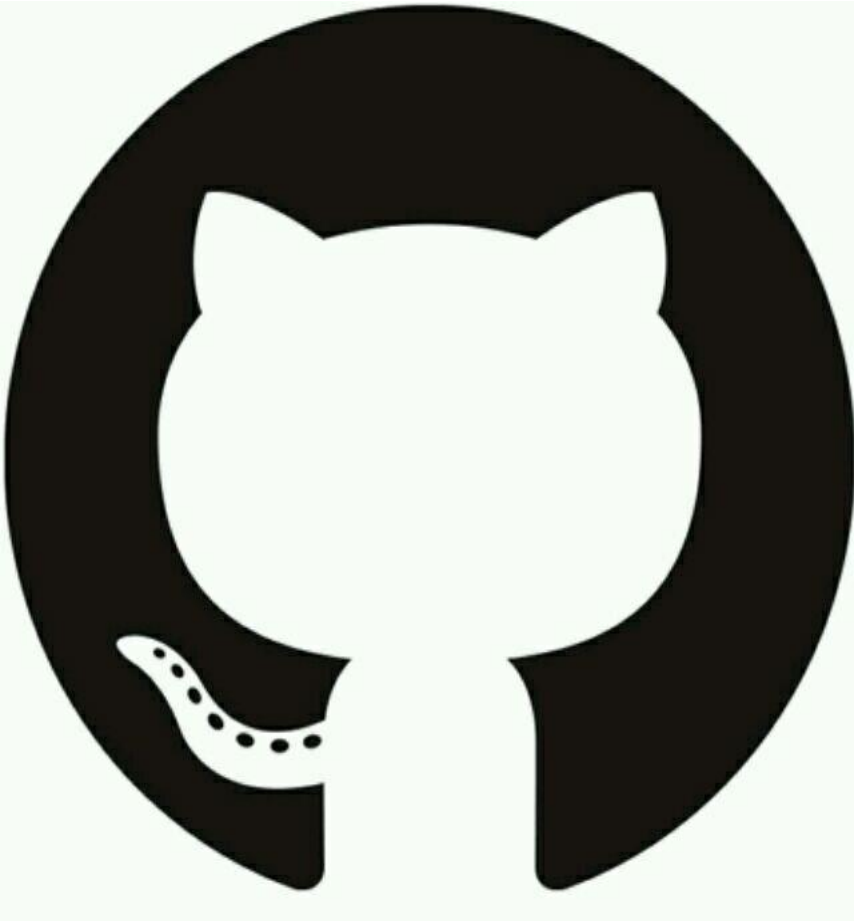


INTRODUCTION

TO

Computer Science

Writed in *2017*

学计算机的同学不想和你说话
并向你扔过来一堆编程语言

对方不想和你说话
并向你扔了一堆BUG

1.Hello world 程序.C

```
#include<stdio.h>          //C 预处理器指令复制标准输入输出头文件并粘贴在这里
int main()           //main()必须是 C 程序开始的函数，返回到操作系统的值是整数
{                                    //花括号把 main()函数体括起来
    printf("hello world\n");         //输出函数，\n 表示换行，语句以分号结尾
    return 0;                        //返回值为 0，建议保留统一代码风格
}
```

2.输入复制到输出.C

```
#include<stdio.h>
int main()
{
    char ch;                          //声明 1 个字符型变量
    while ((ch = getchar()) != EOF)
        putchar(ch);                      //相当于 printf("%c",ch)
}
```

圆括号中的条件表示：变量 c 中包含键盘输入的字符，getchar()是按一次性读一个字符的标准库函数，while 循环使得它能一个接一个字符地读取文本，!=表示不等于，EOF 是文件结束符，它定义在头文件 stdio.h 中，是一个整型数，VS 中用 ctrl+Z 然后 Enter 来表示 EOF ，getchar()和 putchar()都不是真正的函数，它们是供预处理器使用的宏

3.华氏与摄氏温度转换.C

```
#include<stdio.h>
#define min  0              //使用符号常量，#define 名字 替换文本
#define max  300           //避免使用魔法数(magic number)，利于修改
#define  t  20              //定义了转换表的下限,上限,步长
int main(){
   int f;
   for (f = min; f <= max;f=f+t)
       printf("%3d  %6.1f\n",f,(5.0/9.0)*(f-32));
                                //按顺序输出数值，后面变量用逗号隔开
   return 0;
}
```

for 语句是一种循环语句，是 while 语句的推广，其圆括号中包含 3 个部分，每一部分用分号隔开，第一部分是初始化（仅在循环前执行一次），第二部分是控制循环的测试或条件部分（循环控制将对该条件求值，结果为真则执行循环体，本例的循环是输出-走一步-输出-走一步 ••••••，直到条件为假循环终止），第三部分为增加步长部分

转换说明：%d 十进制整数　%6d 至少 6 个字符宽　%f 打印浮点数　%6.2f 至少 6 格宽，小数点后留 2 位　%o 八进制数　%x 十六进制数　%c 字符　%s 字符串　%%百分号本身

计算机中最小的储存单位是位(bit)，可以储存 0 或 1，1 字节(byte)等于 8 位，C 规定一个字节为 char 类型占用的位数，使用 sizeof()运算符可以知道当前系统的指定变量类型所占的字节个数，还可以知道字符数组的元素（存储单元）个数。C 把函数库中相关的函数归为一类，并为每类函数提供一个头文件，strlen()函数包含在<stirng.h>头文件中，它可以获取字符串的长度。sizeof()和 strlen()都使用%u 转换说明，C 中科学计数法可以写成指数浮点型，如-3.224 乘以 10 的负 3 次方记作-3.224e-3，转换说明为%e，浮点型（float）常量中间不能加空格，浮点数与数学中实数的概念相类似。

变量和常量是程序处理的两种基本数据对象。声明语句说明变量的名字及类型，运算符指定将要进行的操作，表达式则把变量与常量组合起来生成新的值。变量名是由字母和数字组成的序列，第一个字符必须为字母，大小写字母是区分的。在传统的 C 用法中，变量名用小写字母，符号常量名全部用大写字母。

一个字符常量是一个整数，书写时将一个字符括在单引号中，字符在机器字符集中的数值就是字符常量的值。

枚举常量是另一种类型的常量，枚举是一个常量整型值的列表，如 enum Boolean{yes,no};

在没有显式说明的情况下，enum 类型中第一个枚举名的值为 0，第二个为 1，依此类推。如果只指定了部分枚举名的值，那么未指定的枚举名的值将依着最后一个指定值向后递增。不同枚举中的名字必须互不相同，同一枚举中不同的名字可以具有相同的值。

如 enum months{JAN=1,FEB,MAR,APR,MAY,JUN,JUL,AUG,SEP,OCT,NOV,DEC};

所有变量都必须先声明后使用，尽管某些变量可以通过上下文隐式地声明。

任何变量的声明都可以使用 const 限定符，它指定变量的值不能被修改，对数组而言，所有元素的值都不能被修改。

运算符可以分为：算术，关系，逻辑，自增自减，按位，赋值等。按操作数的个数可以分为一元，二元，三元。

根据定义，在关系表达式或逻辑表达式中，如果关系为真，则表达式的结果值为 1；如果为假，则结果值为 0。逻辑非运算符! 的作用是将非 0 操作数转换为 0，将操作数 0 转换为 1。该运算符通常用于 if(!valid)而一般不采用 if(valid==0)的形式。

下面按照所有运算符的优先级与结合性排序，同一行中的运算符具有相同的优先级，各行从上往下优先级逐行降低，结合性默认为从左到右。

```
()   []   ->   .
!    ~   ++  --  +  -  *  &  (type)  sizeof      结合性：←
*    /   %
+    -
<<   >>
<    <=  >   >=
==   !=
&
^
|
&&
||
?:                                               结合性：←
=    +=  -=   *=  /=  %=   &=                    结合性：←
^=   |=  <<=  >>=  ,
```

C 没有指定同一运算符中多个操作数的计算顺序，也没有指定函数各参数的求值顺序。在不同的编译器中可能会产生不同的结果，C 标准对这类问题有意未作具体规定。在任何一种编程语言中，如果代码的执行结果与求值顺序相关，则都是不好的程序设计风格。

可以用这样一段代码判断闰年：

```
if((year%4==0&&year%100!=0)||year%400==0)
```

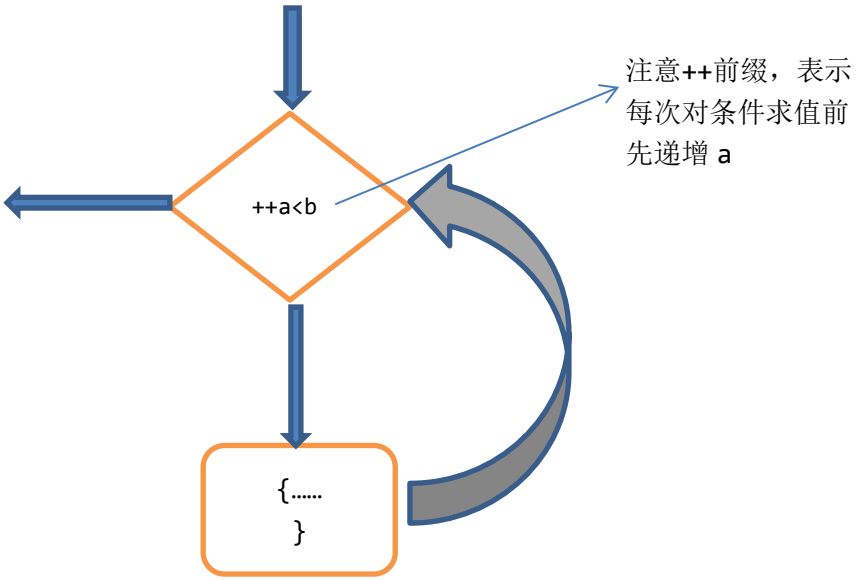
下面这段语句打印一个数组的 n 个元素，每行打印 10 个元素，每列之间用一个空格隔开

```
for(i=0;i<n;i++)
    printf("%6d%c",a[i],(i%10==9||i==n-1) ? '\n' : ' ');
```

4.秒数转化为分秒.C

```
#include<stdio.h>
#define persec 60
int main(){
 int sec, min, left;
 printf("Enter the number of seconds:\n");
 scanf_s("%d", &sec);
//输入整数，Visual Studio 中标准输入函数要加_s，后面对应处加上&
 while (sec > 0) {
     min = sec / persec;            //求商并取整
     left = sec%persec;             //求模运算，表示求出余数
     printf("%d seconds is %d minutes,%d seconds.\n", sec, min, left);
     printf("Enter next value\n");
     scanf_s("%d", &sec);
 }
 printf("done!\n");
 return 0;}
```

while 循环的工作原理：



数组是按顺序储存的一系列类型相同的值，如 float uhuhu[20]；声明了一个含有 20 个元素的名为 uhuhu 的数组，每个元素都可以储存一个 float（浮点）类型的值，第一个元素是 uhuhu[0],元素编号（数组下标）从 0 开始。字符串储存在 char 类型的数组中，如果 char 类型的数组末尾放了一个空字符\0，则该字符数组也是一个字符串。声明语句如下：

```
char jijiji[78]="Good evening,ladies and gentleman";
char huhuhu='o';
```

考虑下面的代码：

```
char words[14];     //声明一个字符数组
fgets(words,9, stdin); //通过第 2 个参数 n 限制读入的字符数（读入 n-1 个），并保留换行符
puts(words);     //输出字符串，并在末尾添加换行
fputs(words,stdout); //输出字符串，不添加换行，stdin 表示标准输入 stdout 表示标准输出
puts("done");
```

输入如下：
asdfghjklz
输出如下：
asdfghjk
asdfghjkdone
请按任意键继续. . .

5.前缀和后缀.C

```
#include<stdio.h>
#define format  "%s! c is cool! \n"
#define format  "%s! c is cool! \n"
int main()
{
 int num = 10;
 printf(format,format); //相当于printf("%s! c is cool!\n","%s! c is cool!\n");
printf("%d",++num);          //先递增，再打印
printf("%d",num++);         //先打印，再递增
printf("%d",num--);        //先打印，再递减
 printf("%d",num);       //打印当前值
 return 0;}
```

输出如下：
%s! c is cool!
! c is cool!
11111211 请按任意键继续. . .

6.求输入整数的和.C

```
#include<stdio.h>
int main(void)
{
 long num;
 long sum = 0L;    //sum初始化为0
 int status;
 printf("please enter an integer to be summed ");
 printf("(q to quit):");
 status = scanf_s("%ld", &num);          //如果scanf成功读取一个整数，就把这个数存给num并返回1，返回值赋给status，这样做同时更新了num和status的值
 while (status == 1)      //如果输入的不是整数，scanf读取失败返回0，循环终止。如果把表示逻辑关系的==改成表示赋值的=，输入字母之后会陷入无限失败循环
 {
     sum = sum + num;
     printf("please enter next integer (q to quit):");
                                 //其实除了q之外其他字母也行
     status = scanf_s("%ld", &num);}
 printf("Those integers sum to %ld.\n", sum);
 return 0;}
```

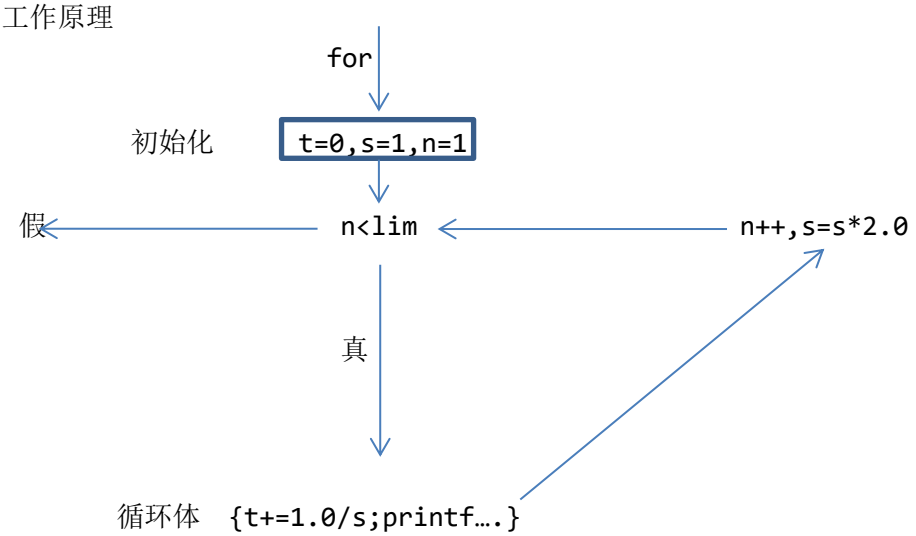
在编程中，表示真或假的变量称为布尔变量，_Bool 类型只能存储 0（假）和 1（真）
C 有三种逻辑运算符，分别是&&（与）、||（或）、!（非）
如果在程序中包含 **iso646.h** 头文件，可以用 and 、or 、not 分别代替
预处理是编译前单独执行的一个步骤，两个最常用的指令是：

文件包含 #include<文件名>
宏替换　#define　名字　替换文本
使用标准库函数时，要注意它是由哪个头文件提供的，使用之前要先在预处理指令中把这个头文件包含

7.芝诺悖论.C

```
#include<stdio.h>
 int main(){
 int n,lim;              //n 表示项计数，lim 表示最大项
 double t, s;
 printf("Enter the number of terms you want:");
 scanf_s("%d", &lim);
 for (t = 0, s = 1, n = 1; n<=lim; n++, s *= 2.0)      //相当于 s=s*2.0
 {
     t += 1.0 / s;           //+=是复合赋值运算符，相当于 t=t+(1.0/s)
     printf("t=%f when terms =%d.\n", t, n);
 }
 return 0;}
```

for 循环的工作原理

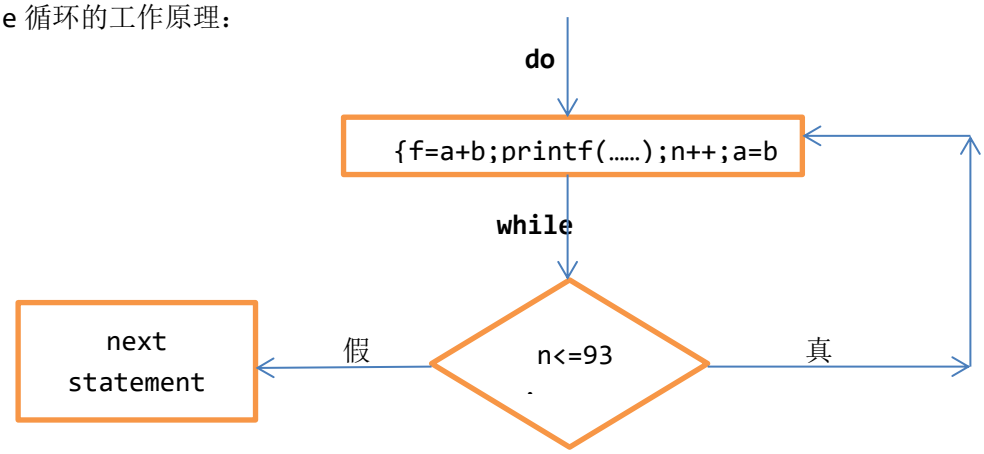


8.打印斐波那契数列前 93 项.C

```
#include<stdio.h>
int main()
{
        unsigned long long a = 0, b = 1, n = 2, f;
                                                //最大范围类型（无符号）
        printf("n   F(n)\n1   1\n");
        do
         {
            f = a + b;
            printf("%llu   %llu\n", n, f);
            n++; a = b; b = f;          //先把 b 的值赋给 a，再把 f 的值赋给 b
         }while (n <= 93);     //do while 循环的特殊之处，while 语句后面加分号
    }                          //第 94 项溢出，最大为 1844…..（20 位）
```

语法糖是一种便捷写法，相当于汉语里的成语，就是把库做到语言的程度，它的存在可以增加代码可读性，简化程序员操作。for 循环就是一个语法糖，它可以改写成等价的 while 循环，但是比 while 循环更加简洁

do while 循环的工作原理：



do while 循环在执行循环体后才执行测试条件，所以至少执行循环体一次，它属于出口条件循环。for 循环和 while 循环是先执行测试条件，属于入口条件循环。

9.嵌套循环.C

```
#include<stdio.h>
#define ROWS 6                  //行数
#define CHARS 6                  //列数
int main()
{
 int a,b;
 for (b = 0; b < ROWS; b++)            //外层循环
 {
     for (a=0; a<CHARS; a++)         //内层循环
         printf("$");               //内层循环体
     printf("\n");                 //外层循环体
 }
 return 0;  }
```

输出如下：　　把 CHARS 改成 b,则输出变为：

```
$$$$$$
$$$$$$                $
$$$$$$  ──────→       $$
$$$$$$                $$$
$$$$$$                $$$$
$$$$$$                $$$$$
```

什么程序会输出下面的结果？做了什么改动？

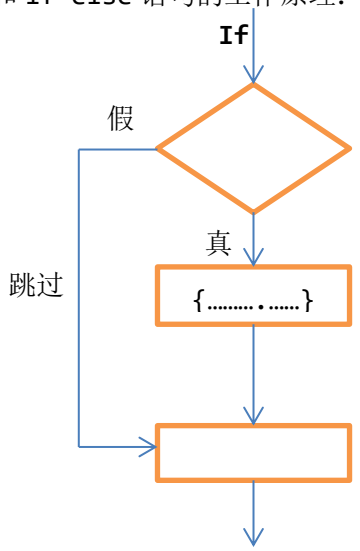
```
ABCDEF                ABCDEF
ABCDEF                BCDEF
ABCDEF                CDEF
ABCDEF                DEF
ABCDEF                EF
ABCDEF                F
```

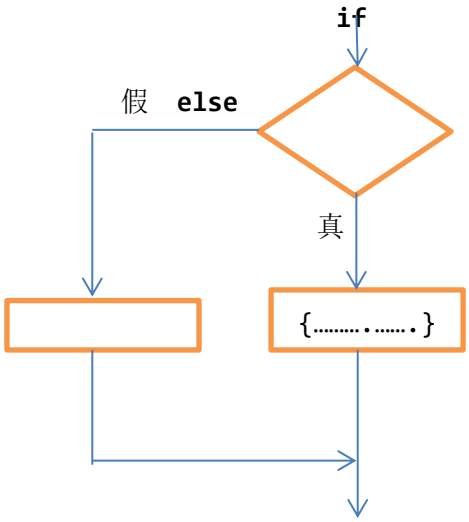
是这样的一个程序：

```
#include<stdio.h>
#define ROWS 6
#define CHARS 6
int main(){
 int a;
 char c;
```

```
    for (a = 0; a < ROWS; a++){
        for (c = 'A'; c < ('A' + CHARS); c++)
                                //将'A'改为（'A'+a）,c++表示按 ASCII 顺序
            printf("%c", c);
        printf("\n");}
    return 0;}
```

if 和 if else 语句的工作原理：





如果没有花括号，else 与它上方最近的 if 配对，否则跳过花括号

else if 语句可以实现多重选择

```
    if (score < 1000)   bonus = 0;        //（—∞，1000）
    else if (score < 1500)   bonus = 1;   // [1000,1500)
    else if (score < 2000)   bonus = 2;   //[1500,2000)
    else if (score < 2500)   bonus = 4;    //[2000,2500)
    else       bonus = 6;                    //[2500,+∞)  分段函数
```

C 提供条件表达式作为表达 if else 语句的一种便携方式，条件运算符是 C 中唯一的三元运算符，其形式是：expression1 ? expression2 : expression3

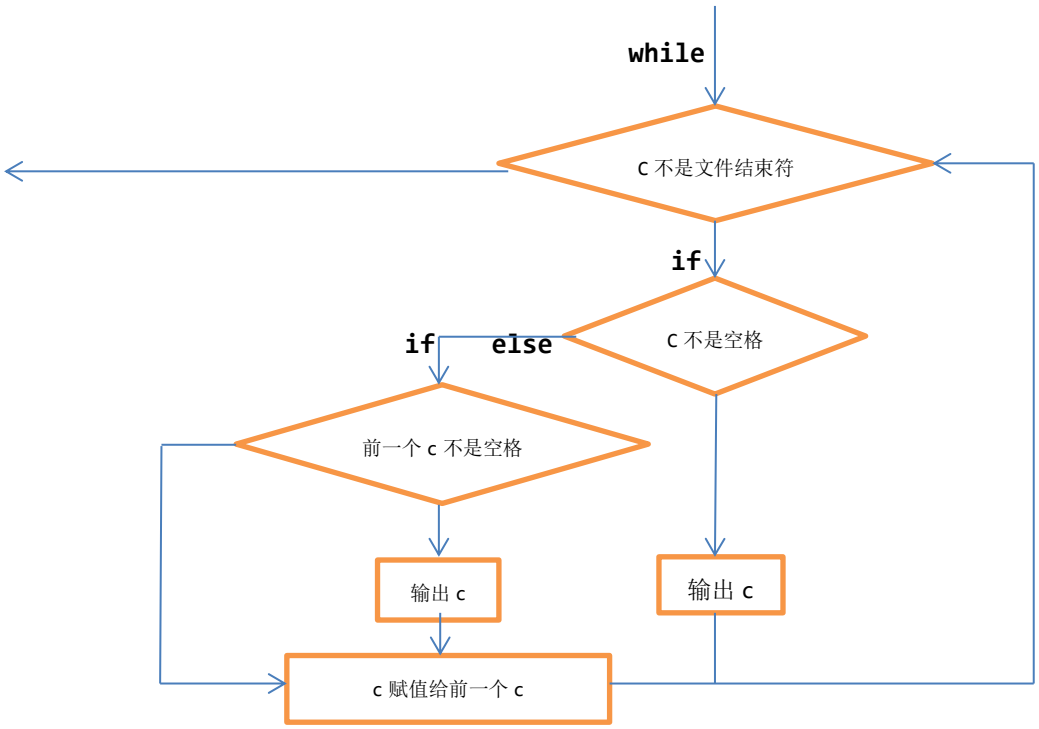
如果 expression1 为真(非 0)，整个表达式的值为 expression2，否则(为假，0)为 expression3

定义是指给变量分配确定的存储单元，声明是指说明变量的类型。作用域描述程序中可访问标识符的区域。块是用一对花括号括起来的代码区域。存储期是指给变量分配的内存保留了多长时间。在函数内部或块内定义的变量叫局部（内部）变量，函数的形参也是内部变量；在函数外部定义的变量叫全局（外部）变量。C 有 4 种存储期：自动、静态、动态分配、线程。局部变量可以声明为自动或静态，全局变量只能是静态。当全局变量定义在后，引用它的函数在前时，应该在引用的地方用 extern 对此全局变量进行说明

存储类别	存储期	作用域	链接	声明方式
自动	自动	块	无	块内，auto
寄存器	自动	块	无	块内，register
静态外部链接	静态	文件	外部	所有函数外
静态内部链接	静态	文件	内部	函数外 static
静态无链接	静态	块	无	块内，static

10.输入一行复制到输出，并将连续的多个空格改成一个空格.C

算法分析：用一个 lastc 记录前一个字符，把它初始化为一个任意的非空格字符，每次循环的最后把它刷新




```
#include<stdio.h>
#define NONBLANK 'a'
void main()
{
    char c, lastc;
    lastc = NONBLANK;
    while ((c = getchar()) != EOF)
    {
        if (c != ' ')
            putchar(c);
        else if (lastc != ' ')
            putchar(c);
        lastc = c;
      }
}
```

其中的循环体可以改写成：

```
if (c != ' ' || lastc != ' ')
     putchar(c);
lastc = c;
```

单词：一个不含空白字符的字符序列；空白字符（单词分隔符）=空格+换行+制表

使用 **ctype.h** 头文件中的函数 isspace()可以判断某字符是不是空白字符，如果是，可以把 isspace(c)设为真(0)，否则为 1。

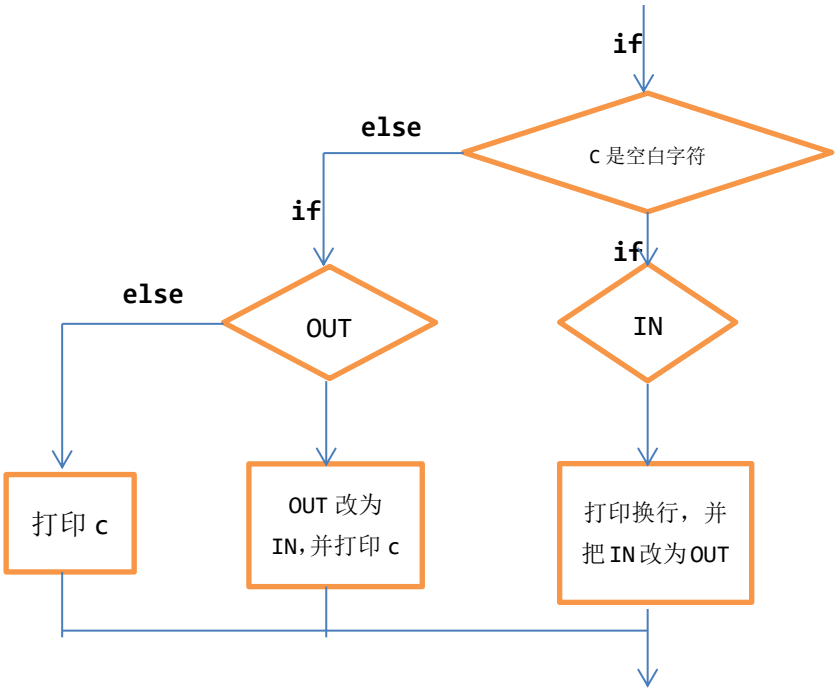
使用 OUT 和 IN 表示当前字符在单词外部或单词内部

		H	E	L	L	O		
0	0	1	1	1	1	1	0	0
OUT	OUT	OUT	IN	IN	IN	IN	IN	OUT

开头　　结尾

11.以每行一个单词的形式打印输入.C

算法的原理：可以把循环体的工作原理设计成下图这样




```
#include<stdio.h>
#define OUT 0
#define IN 1
void main()
{
    char c;
    int state;
    state = OUT;
    while ((c = getchar()) != EOF)
    {
        if (c == ' ' || c == '\n' || c == '\t')
        {
            if (state == IN)
            {
                putchar('\n');
                state = OUT;
            }
        }
        else if (state ==OUT)
        {
            state = IN;
            putchar(c);
        }
        else
            putchar(c);
    }
}
```

12.统计字符数、单词数、行数.C

```
#include<stdio.h>
#define IN 1            //在单词内
#define OUT 0          //在单词外
#define STOP '|'         //结束符
```

```
int main()
{
    char c;
    int nl, nw, nc, state;
    state = OUT;      //用 state 变量记录当前字符在单词内还是外，初始化在外
    nl = nw = nc = 0;
    printf("Enter text to be analyzed( | to terminate):\n");
    while ((c = getchar()) != STOP)
    {
        ++nc;                   //统计字符
        if (c == '\n')
            ++nl;               //统计行
        if (c == ' ' || c == '\n' || c == '\t')
                                //如果是空格或换行符或制表符（空白字符）
            state = OUT;            //记录状态为在外
        else if (state == OUT)
        {
                                //如果不是空白字符，且在外，说明是单词的开头
            state = IN;             //手动把状态从在外修改为在内
            ++nw;           //只有在状态从外设置为内时，递增单词计数
        }
    }
    printf("characters=%d,words=%d,lines=%d\n", nc, nw, nl);
}
```

在 C 中可以简单、方便、高效地使用**函数**，除了函数库提供的函数外，我们可以自己编写一些函数，定义一个函数之后就可以在主程序中调用它，这样使得代码清晰易读，函数定义可以以任意次序出现在一个源文件或多个源文件中

13.编写一个简单的求幂函数.C

```
#include<stdio.h>
int power(int m, int n);     //函数原型，也可以写成 int power(int,int)
void main(){
    int i;
    for (i = 1; i <=10; ++i)
     printf("%-2d|%-5d|%-7d|%-9d\n",i,power(2,i),power(3,i),power(5,i));
}
int power(int base, int n){      //返回值类型 函数名(0 个或多个参数声明){
    int i, p=1;              //声明部分
    for (i = 1; i <= n; ++i)   //语句序列
        p = p*base;
    return p;}              //return 表达式;}
```

我们通常把函数定义中圆括号内列表中出现的变量称为形式参数，而把函数调用中与形式参数对应的值称为实际参数。函数原型是一个出现在 main()之前的声明语句，它必须与函数的定义和用法一致，函数原型与定义中的参数名不要求相同

宏替换中的替换文本可以是任意的，宏定义也可以带参数，且参数可以是任何数据类型

```
#define  forever  for(;;)     //无限循环
#define  max(a,b)  ((a)>(b)?(a):(b))  //调用宏时直接将替换文本插入到代码中
```

可以使用#undef 指令取消名字的宏定义，名字的作用域到此结束，如#undef forever

14.打印输入中的最长文本行.C

```
#include<stdio.h>
#define MAXLINE 1000              //允许的输入行的最大长度
int getline(char line[], int maxline);
```

```
                        //定义的函数放在main()前面以供调用，后面加分号
void copy(char to[], char from[]);
int main()
{
    int len, max=0;   //len表示当前行长度，max表示目前为止发现的最长行的长度
    char line[MAXLINE], longest[MAXLINE];
                              //line表示当前行，longest用于保存最长行
    while ((len = getline(line,MAXLINE))>0)    //测试条件：还有未处理的行
        if (len > max)
        {                             //测试条件：该行比已处理的最长行还要长
            max = len;                                //该行变成新的最长行
            copy(longest, line);
        }                                        //保存最长行
    if (max > 0)          //测试条件：存在这样的最长行
        printf("%s", longest);       //打印最长行
    return 0;
} //按这种逻辑，如果输入中同样长且最长的行有多个，则只打印这些行中最上面那一行
int getline(char s[], int lim)
//getline函数：将一行读入到s中并返回该行的长度
{ //main与getline通过2个参数s（声明为数组）和lim以及1个返回值进行数据交换
    int  i;            //函数内部声明的局部变量，只在调用并执行函数时存在
    char c;
    for (i = 0; i < lim - 1 && (c = getchar()) != EOF&&c != '\n'; ++i)
        s[i] = c;          //字符一个接一个的放在s中
    if (c == '\n')
    {
        s[i] = c;          //倒数第二个格子里面放换行符
        ++i;
    }
    s[i] = '\0';         //数组的最后一个格子里面放空字符，表示字符串的结束
    return i;
}
void copy(char to[], char from[])
{                                     //copy函数：将from复制到to
    int i = 0;      //这个i和上面那个i没有关系，它们是函数各自声明的局部变量
    while ((to[i] = from[i]) != '\0')       //使用循环来实现一个接一个的复制
        ++i;
}                  //copy仅用于执行复制动作，返回类型void说明它不返回任何值
```

使用fgets函数很容易实现getline函数：

```
/*getline函数：读入一个输入行并返回其长度*/
int getline(char *line,int max){
    if(fgets(line,max,stdin)==NULL)
       return 0;
    else
       return strlen(line);}
```

接下来介绍位操作，1字节是8位，可以从左到右给这8位依次编号为7~0，为了方便理解，我们用二进制写出值，但是在实际的程序中并不是这样，用一般形式的整型变量或常量即可。

~　二进制反码（一元运算符）:1变为0，0变为1

&　按位与（二元运算符）：两个操作数相应位都为1时，结果才为1

|　按位或：都为1或其中一个为1，结果为1

^　按位异或：只有其中一个为1时，结果才为1，否则为0

(　)<<n　左移运算符：所有位左移 n 格，用 0 填充空出的位置，丢掉移出的位，相当于乘以 2 的 n 次幂（类似于在十进制中移动小数点）

(　)>>n　右移运算符：所有位右移 n 格，用 0 填充空出的位置，丢掉移出的位，相当于除以 2 的 n 次幂

C 中的**控制流**语句用于控制各计算操作执行的次序，前面我们介绍的用于多路判定的 if 语句：if(表达式) 语句 1

```
    else if(表达式)  语句 2  ……
    else    语句 n
```

其中的各表达式将被依次求值，一旦某个表达式结果为真，则执行与之相关的语句，并结束整个语句序列的执行。其中各语句既可以是单条语句，也可以是花括号内的复合语句。最后一个 else 部分用于处理“以上条件均不成立”的情况，该部分也可以用来检查错误，以捕获“不可能”的条件。

break 语句用于跳出循环或跳出 switch 语句，continue 语句用于开始下一次循环的执行

```
while(…)
{
  ……
  break;
  ……
}
next statement
```

```
while(…)
{
 ……
   continue;
 ……
 }
   next statement
```

switch 语句是一种多路判定语句，其中“case 常量：”是一个标签，标签+后面的语句=一个分支，每一个分支都由一个或多个整数值常量或常量表达式标记，所有的分支构成列表，各分支的排列顺序是任意的。程序执行首先对表达式求值，然后扫描列表，如果某个分支与表达式的值匹配，则从该分支开始执行，如果没有一个匹配的，就执行 default 分支，break 语句可以让程序直接跳出 switch 语句而执行下一条语句

```
switch(整型表达式)
{
    case 常量 1:statement1
         break;
    case 匹配常量：statement2
    case 常量 3：statement3
         break;
    case 常量 4：statement4
    default:
         last statement
}
    next statement
```

15.统计数字、空白字符及其他字符.C

```
#include<stdio.h>
int main()
{
      char c;
      int i, nwhite, nother, ndigit[10];
      nwhite = nother = 0;
      for (i = 0; i < 10; i++)          //用 for 循环初始化数组
```

```
        ndigit[i] = 0;
      while ((c = getchar()) != EOF) {
        switch (c)
        {
        case '0':case'1':case'2':case'3':case'4':
        case '5':case'6':case'7':case'8':case'9':
            ndigit[c - '0']++;
            break;
        case' ':case'\n':case'\t':
            nwhite++;
            break;
        default:
            nother++;
            break;
        }}
      printf("digits=");
      for (i = 0; i < 10; i++)
         printf("%d\t", ndigit[i]);
      printf(",white space=%d,other=%d\n", nwhite, nother);
      return 0;
}
```

16. 删除字符串中的字母v.C

```
#include<stdio.h>
void squeeze(char line[], char);
int main(){
      char line[123], po='v';
      scanf_s("%s", line);
      squeeze(line,po);                        //就这样调用函数
      printf("%s", line);
}
void squeeze(char s[], char c){
      int i, j;
      for (i = j = 0; s[i] != '\0'; i++)
         if (s[i] != c)                  //每当出现一个不是 c 的字符时
             s[j++] = s[i];
               //把它拷贝到数组中下标为 j 的位置，然后 j 增加 1 以处理下一个
      s[j] = '\0';
}                              //循环结束，尾部放一个空字符表示数组结尾
```

17.输出以下图案.C

```
*
**
***
****
*****
****
***
**
*
#include<stdio.h>
void shit(int stars)
{                                          //这是最糟糕的函数名
      while (stars-- > 0)
         printf("*");
```

```
      printf("\n");
}
void fuck(int stars)
{                                       //但也是一个递归的巧妙应用
      if (stars == 5)
         shit(stars);
      else
      {
         shit(stars);
         fuck(stars+1);
         shit(stars);
      }
}
int main()
{
      fuck(1);
      return 0;
}
```

18.倒置字符串.C

```
#include<stdio.h>
#include<string.h>
void reverse(char a[]);
main()
{
    char a[123];
    scanf_s("%s",a);
    reverse(a);
    printf("%s",a);
}
void reverse(char s[])
{
    int c,i,j;
    for(i=0,j=strlen(s)-1;i<j;i++,j--)
    {
                            //在字符串正中间放一个镜子，依次调换镜子两边的字符
        c=s[i];             //交换两个瓶子中的饮料需要第 3 个空瓶子
        s[i]=s[j];
        s[j]=c;
    }
}
```

19.连接两个字符串.C

```
#include<stdio.h>
void strcat(char a[], char b[]);
int main()
{
      char a[1000], b[123];
      scanf_s("%s", a);
      printf("Please enter another array\n");
      scanf_s("%s", b);
      strcat(a, b);                     //把 b 接到 a 的尾巴上
      printf("%s", a);
}
void strcat(char s[], char t[]){
      int i=0, j=0;                 //分别用 i 和 j 储存数组 s 和 t 的长度
      while (s[i] != '\0')                    //找到 s 数组的尾部
```

```
        i++;
      while ((s[i++] = t[j++]) != '\0');
                    //循环体是个分号，不执行动作，拷贝在测试条件中完成
}
```

20.数组的shell 排序算法.C

```
#include<stdio.h>
void sort(int v[],int);
main()
{
    int me[11]={15,27,11,36,31,59,20,23,14,33,28,};
    sort(me,11);
    for(int i=0;i<11;i++)
        printf("%d %5d\n",i,me[i]);              //for 循环打印数组
}
void sort(int v[],int n)
{
    int gap,i,j,temp;
    for(gap=n/2;gap>0;gap/=2)
                        //2 个被比较元素之间的距离从 n/2 开始对折，直到它们重合
        for(i=gap;i<n;i++)                    //在元素间移动位置
            for(j=i-gap;j>=0&&v[j]>v[j+gap];j-=gap)
            {               //比较各对相距 gap 个位置的元素，如果逆序

                temp=v[j];                //就把它们调换过来
                v[j]=v[j+gap];
                v[j+gap]=temp;
            }
}
```

可以通过按位运算符来计算各种类型变量的取值范围，比如要确定 signed int 类型的取值范围，先用~0 把数字 0 的各个二进制位全部转换为 1，然后将结果值转换为 unsigned int 类型，再把这个无符号类型值右移一位（>>1）以清除符号位，最后再把它转为 int 类型。这一系列操作最终结果就是 signed int 类型的最大值，如果要最小值就在前面加一个负号。如果要确定无符号类型的变量取值范围，就不需要用到右移运算符。

21.编写一个程序确定各种类型变量的取值范围.C

```
#include<stdio.h>
void main(){
      printf("signed short min=%d\n", -(short)((unsigned short)~0 >> 1));
      printf("unsigned long max=%lu\n", (unsigned long)~0);}
```

C 允许函数调用它自己，这叫**递归**，结束递归是使用递归的难点，递归方案更简洁但效率没有循环高

22.函数递归演示.C

```
#include<stdio.h>
void newton(int);
void main(){ newton(1); }
void newton(int n)
{
      printf("level %d: n location %p\n", n, &n);
                                        //%p 和&n 组合打印变量 n 的内存地址
      if (n < 4)
         newton(n + 1);
      printf("LEVEL %d: n lacation %p\n", n, &n);
}
```

输出如下：

```
level 1: n location 0133F76C
level 2: n location 0133F694
level 3: n location 0133F5BC
level 4: n location 0133F4E4
LEVEL 4: n lacation 0133F4E4
LEVEL 3: n lacation 0133F5BC
LEVEL 2: n lacation 0133F694
LEVEL 1: n lacation 0133F76C
```

每级函数调用都有自己的变量，每个变量名都是 n，但是它们的值各不相同。每次函数调用都会返回一次，执行完毕时控制权将被传回上一级递归。递归函数中位于递归调用之前的语句，均按被调函数的顺序执行；而位于递归调用之后的语句则按被调函数相反的顺序执行。这种特性在解决涉及相反顺序的问题时很有用。

23.以二进制形式打印整数.C

```
#include<stdio.h>
void binary(unsigned long);
void main()
{
      unsigned long number;
      while (scanf_s("%lu", &number) == 1)
      {
          binary(number);
          putchar('\n');
      }
}
void binary(unsigned long n)
{
      int r ;
      r = n % 2;
      if (n >= 2)
          binary(n / 2);
      putchar(r == 0 ? '0' : '1');
}
```

算法分析：在二进制中，奇数的末尾一定是 1，偶数的末尾一定是 0。一般而言，对于数字 n，其二进制的最后一位是 n%2，因此，计算的第一位数字实际上是待输出二进制数的最后一位，这一规律提醒我们，在递归函数调用之前计算 n%2，在递归函数调用之后打印计算结果，这样，计算的第 1 个值正好是最后一个打印的值。要获得下一位数字，必须把原数除以 2，这种算法相当于在十进制下把小数点左移一位。那么，程序该何时停止计算？当与 2 相除的结果小于 2 时停止，因为只要结果大于或等于 2 就说明还有二进制位，每次除以 2 就相当于去掉一个二进制位，直到计算出最后一位。

24.画圣诞树.C

```
#include<stdio.h>
#include<stdlib.h>
int main(int argc,char*argv[])
{int n=argc>1?atoi(argv[1]):4;
for(int j=1;j<=n;j++){
    int s=1<<j,k=(1<<n)-s,x;
    for(int y=s-j;y>=0;y--,putchar('\n')){
        for(x=0;x<y+k;x++)
            printf(" ");
        for(x=0;x+y<s;x++)
            printf("%c",'!'^y&x);
```

```
        for(x=1;x+y<s;x++)
            printf("%c",'!'^y&(s-y-x-1));}}}
```

25.画心形.C

```
#include<stdio.h>
int main(){
      for (float y = 1.5f; y > -1.5f; y -= 0.1f){
        for (float x = -1.5f; x < 1.5f; x += 0.05f){
            float a = x*x + y*y - 1;
            putchar(a*a*a - x*x*y*y*y <= 0.0f ? '*' : ' ');}
        putchar('\n');}}
```

26.将输入的一行字符串中除 t 以外的其他小写字母变成大写字母.C

```
int toupper(int ch);
int getchar();void putchar(int);/*可能出现 warning:putchar 函数类型冲突*/
int main();
void nothing(char c){}
int self(int c){ return c; }
void doit(char c){
      int(*fuck[2])(int) = { toupper, self };
      putchar(fuck[c == 't'](c));
      main();}
int main(){
      void(*fuck[2])(char) = { doit, nothing };
      char ch;
      ch = getchar();
      fuck[ch == '\n'](ch);}
```

现在我们开始学习 C 语言的精髓：**指针**

对于一个储物柜，如果你在一张纸条上写上 B107，然后把这张纸条放进 K467 号柜子里面，那么我们就说 k467 号柜是指向 B107 号柜的指针

可以这样理解指针的作用，假如交货的时候小明随机把毒品放在 C300 到 C320 柜中，然后把线索放在 C322 中，小强只要先去 C322 里找线索，然后取毒品就可以了，小明和小强不用直接接触

所有高级语言里，C 的指针是操纵硬件的唯一方式。C 只有值的传递，无法直接传递引用，要想传递引用必须通过指针间接实现。实际上指针这概念在其他高级语言中也有运用，只是人家不叫指针而叫引用。如果用好指针，可以发挥 C 的强大威力。数据结构依靠的就是指针，不能靠内存的绝对位置表示数据的关系吧？C 的好处就是：没有多余的抽象/封装，一切以硬件界面为准，数据就是数据，内存就是内存，封装成类反而棘手多了，只要是硬件允许的都可以做，而所有这些都是围绕着指针实现的。

指针是一种保存变量地址的变量，使用指针可以生成更高效紧凑的代码。

一元运算符&可用于取一个对象的地址。一元运算符*是间接寻址或间接引用运算符，当它作用于指针时将访问指针所指向的对象。

```
      int x = 1, y = 2, z[10];
      int *p;          //p 是指向 int 类型的指针
      p = &x;          //p 现在指向 x
      y = *p;           //y 的值现在为 1
      *p = 0;           //x 的值现在为 0
      p = &z[6];            //p 现在指向 z[6]
```

编写程序时，可以认为变量有 2 个属性：名称和值，计算机编译和加载程序后，也认为变量有 2 个属性：地址和值，地址就是变量在计算机内部的名称。在许多高级语言中，地址都归计算机管，对程序员隐藏，然而在 C 中，可以通过&运算符访问地址，通过*运算符获得储

存在地址上的值。So，普通变量把值作为基本量，把地址作为通过&运算符得到的派生量，而指针变量把地址作为基本量，把值作为通过*运算符得到的派生量。

27.数组元素之和.C

```
#include<stdio.h>
#define size 10
int sump(int *start, int *end);
void main(){
      int marbles[size] = { 28, 23, 13, 6, 2, 37, 10, 26, 17, 789 };
      long answer;                //start 指针指向数组头部 end 指向尾部
      answer = sump(marbles, marbles + size);
      printf("the total number of marbles is %d.\n", answer);}
int sump(int *start, int *end){                  //使用指针形参
      int total = 0;
      while (start < end)
         total += *start++;
                          //把元素的值加到 total 上，再让指针指向下一个元素
      return total;}
```

数组名就是数组首元素的地址，即 marbles==&marbles[0];在 C 中，指针+1 的意思是增加一个存储单元，对数组而言是下一个元素的地址，而不是下一个字节的地址，这是为什么必须声明指针所指向对象类型的原因之一。有效的指针运算只有 4 种：相同类型指针之间的赋值运算，指针和整数的加减运算，指向相同数组中元素的两个指针间的减法和关系比较运算，以及将指针赋值为 0 或指针与 0 之间的比较运算。

二维数组就是数组的数组，例如语句 float rain[5][12];声明了一个数组 rain，它有 5 个元素，每一个元素是一个包含 12 个 float 类型数的数组，rain 有 5 行 12 列

28.画科赫雪花.C

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
static char* image;
static int x, y, w, h;
static void put(int i, int j, char c){
      char *p = image + j*w + i;
      if (*p == ' ' || *p == '_')
        *p = c;}
static void kochcurve(int n, int dir){
      if (n==0)
        switch (dir % 6){
        case 0:put(x++, y , '_');put(x++, y, '_');break;
        case 1:put(x++, y--, '/');break;
        case 2:put(--x, y--, '\\');break;
        case 3:put(--x, y, '_'); put(--x, y, '_');break;
        case 4:put(--x, ++y, '/');break;
        case 5:put(x++, ++y, '\\');break;}
      else{
        kochcurve(n - 1, dir);
        kochcurve(n - 1, dir + 1);
        kochcurve(n - 1, dir + 5);
        kochcurve(n - 1, dir);}}
int main(){
      w = 2; h = 4;int i,j,n;
      for (n = 0; n < 4; n++){
        image = (char*)malloc(w*h);
        memset(image, ' ', w*h);
```

```
        x = 0; y = h / 4 - 1;
        kochcurve(n, 0);
        kochcurve(n, 4);
        kochcurve(n, 2);
        printf("\nn=%d\n\n", n);
        for (j = 0; j < h; j++){
            for (i = 0; i < w; i++)
                putchar(image[j*w + i]);
            putchar('\n');}
        w *= 3;
        h = n == 0 ? 4 : h * 3;
        free(image);}}
```

29.画三维形状的球体.C

```
#include<stdio.h>
#include<math.h>
int main(){
      float x, y;
      for (y = 1; y >= -1; y -= 0.05f, putchar('\n'))
         for (x = -1; x <= 1; x += 0.025f)
             putchar(x*x + y*y > 1 ? 'M' : "@@%#*+=;:. "[(int)(((x + y + sqrt(1
- (x*x + y*y)))*0.5773502692f + 1)*5.0f +0.5f)]);}
```

30.计算机图形学中一种合成图像的基本代码模板.C

```
#include<stdio.h>
const int w = 20;                    //显示的宽度
const int h = 9;                     //显示的高度
int f(int x, int y)
{
      return triangle(x-11,y)&&triangle((x+y/3*3+3)%6-2,y%3);
}                          //关键部分。表示函数边界，其他代码都是模板
int main(){
      int x, y;
      for (y = 0; y < h; y++)
      {
         for (x = 0; x < w; x++)
             printf(f(x, y) ? "*" : " ");
         puts(" ");
      }
}
int triangle(int x, int y)
{                                 //画等腰直角三角形的函数
      return abs(x) <= y;
}
```

二维数组元素 Zippo[2][1]等价的指针表示法是*(*(Zippo+2)+1)

Zippo　　二维数组首元素（第 1 个一维数组）的地址

Zippo+2　　二维数组的第 3 个元素的地址

*(Zippo+2)　　二维数组的第 3 个元素（即一维数组）的首元素的地址

*(Zippo+2)+1　　二维数组的第 3 个元素的第 2 个元素的地址

((Zippo+2)+1) 二维数组的第 3 个元素的第 2 个元素的即二维数组第 3 行第 2 列的值

C 常常因为声明的语法问题而受到人们的批评，特别是涉及到指针的复杂声明

数组指针：int (*p)[2];　p 指向一个内含 2 个 int 类型值的数组

当且仅当 p 是一个函数的形参时，可以等价声明为 void somefunction(int p[][2]);

指针数组：int *p[2];　p 是包含 2 个元素的数组，每个元素都是指向 int 类型的指针

一般而言，声明一个指向多维数组的指针时，只能省略最左边方括号中的值，因为第一对方括号只用于表示这是一个指针，如 int sum4d(int ar[][12][20][30], int rows);
函数指针：int(*f)(char *, int *);f 是一个指向函数的指针，这个函数返回 int 类型
作为函数形参时，char s[];和 char *s;是等价的，一个通过数组和下标实现的表达式也可以通过指针和偏移量来实现，函数原型中名字 s 可以省略。C 不能把整个数组作为参数传递给函数，但是可以传递数组的地址，然后函数可以使用地址操控原始数组，无论使用数组表示法还是指针表示法，实际上使用的都是指针变量。
指针函数：int *g(char *, int *); g 是一个函数，它返回一个指向 int 类型的指针
scanf()的所有参数都必须是指针

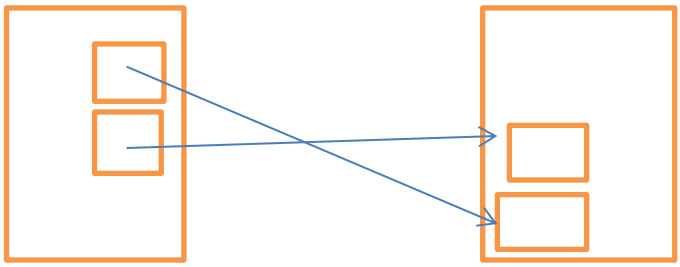
31.日期转换.C

```
#include<stdio.h>
int day_of_year(int year,int month,int day);
void month_day(int year,int yearday,int *pmonth,int *pday);
int main()
{
    int dayofyear,m,d;
    dayofyear=day_of_year(2015,6,7);                //在这里修改你要的值
    month_day(2015,158,&m,&d);        //后面是 2 个地址，存放了 2 个变量 m 和 d
    printf("%d\n%d %d",dayofyear,m,d);
}
static char daytab[2][13]={
            //必须在函数外部声明这个数组，这样 2 个函数就都可以使用该数组
    {0,31,28,31,30,31,30,31,31,30,31,30,31},
    {0,31,29,31,30,31,30,31,31,30,31,30,31}
};//将每月天数分别存放在一个二维数组的两行中比在计算过程中判断 2 月有多少天方便
//day_of_year 函数：将某月某日的日期表示形式转换为某年中第几天
int day_of_year(int year,int month,int day)
{
    int leap;                       //leap 声明为整型，实际上是 bool 型
    char *p;                            //p 是指向数组元素的指针
    leap=year%4==0&&year%100!=0||year%400==0;
                                    //判断闰年的逻辑表达式的值只能是 0 或 1
    p=daytab[leap];            //现在 p 指向一维数组 daytab[leap]的首元素
    while(--month)
     //相当于(--month>0),只需判断条件表达式的值是否为 0 时，比较运算符可以省略
        day+=*++p;
        //先把 p 的指向移动到下一个元素，再取得该元素的值，再把该值加到 day 上
    return day;
 //只要把 day 所在月之前所有月的天数（存放在数组中）和 day 相加，就得到了总天数
}
//month_day 函数：将某年中第几天的日期转换为某月某日
void month_day(int year,int yearday,int *pmonth,int *pday)
{//要得到 2 个值，但 C 不能直接传值，只能通过指针间接引用，所以后面的参数放 2 个地址，用 2 个指针指向它们
    int leap;
    char *p;
    leap=year%4==0&&year%100!=0||year%400==0;
    p=daytab[leap];
    while(yearday>*++p)        //当 yearday 始终大于 p 指向的前 n 项元素的值的和
        yearday-=*p;   //每一次循环就把 yearday 减掉 p 目前指向的月份天数
    *pmonth=p-*(daytab+leap);
            //后者表示 p 指向的一维数组的首元素地址，2 个地址相减就得到月数
```

```
    *pday=yearday;              //循环结束（减完了）就能得到当月中的天数
}
```

被调用函数不能直接修改主调函数中变量的值，参数传递采用传值形式。我们可以使主调程序将指向所要交换的变量传递给被调用函数，由于一元运算符&用来取变量的地址，这样&a就是一个指向变量 a 的指针。swap 函数的所有参数都声明为指针，并且通过这些指针来间接访问它们指向的操作数。

```
swap(&a,&b);
void swap(int *px,int *py)
{
    int temp;
    temp=*px;
    *px=*py;
    *py=temp;
}
```



swap 函数 **main 函数**

C.A.R.Hoare 于 1962 年设计的快速排序算法：对于一个给定的数组，从中选择一个元素，以该元素为界将其余元素划分为 2 个子集，一个子集中的所有元素都小于该元素，另一个子集中的所有元素都大于或等于该元素。对这样的 2 个子集递归执行这一过程，当某个子集中的元素数小于 2 时，这个子集就不需要再次排序，终止递归。下面的代码在每次划分子集时，总是选取各个子数组的中间元素。

```
void qsort(char *v[], int left, int right){
        int i, last;
        void swap(char *v[], int i, int j);
        if (left >= right)
            return;
        swap(v, left, (left + right) / 2);
        last = left;
        for (i = left + 1; i <= right; i++)
            if (v[i]<v[left])
                swap(v, ++last, i);
        swap(v, left, last);
        qsort(v, left, last - 1);
        qsort(v, last + 1, right);
}
void swap(char *v[], int i, int j){
        char *temp;
        temp = v[i];
        v[i] = v[j];
        v[j] = temp;
}
```

库<string.h>中提供的 strcpy 函数：将指针 t 指向的字符串复制到指针 s 指向的位置

```
void strcpy(char *s, char *t){ while (*s++ = *t++); }
```

C 动态分配内存的函数有 4 个：malloc,calloc,free,realloc，它们属于头文件<stdlib.h>

```
double *ptd;
ptd=(double *) malloc(30*sizeof(double));
```

以上代码为 30 个 double 类型值请求内存空间，并设置 ptd 指向该位置。相当于声明一个指针，调用 malloc()并将其返回值赋给（经过强制转换）指针，使用指针访问数组元素。用过 calloc()和 malloc()之后，用 free(ptd);释放 ptd 所指向的内存空间。

结构体是一个或多个变量的集合，这些变量可能为不同类型，为了处理的方便而将这些变量

组织在一个名字之下。
结构声明描述了一个结构的组织布局，类似于 C++中的模板。

```
struct book{                //关键字 struct 后面紧跟一个结构标记 book
    char title[maxtitl];       //结构成员列表，每个成员用自己的声明来描述
    char author[maxautl];//成员可以是任何一种数据类型，甚至其它结构（嵌套结构）
    float value;            //每个成员的声明语句末尾都要加分号
};                        //花括号最后的分号不可少
```

创建结构变量：

```
struct book blood, library;
```

这条语句把 library 和 blood 声明为 2 个使用 book 结构布局的结构体，编译器使用 book 模板为 library 和 blood 分配存储空间。上面两段代码可以合为一段：

```
struct book{                //可以去掉结构标记
  …;  …;  …;  } blood, library;
```

结构的初始化：

```
struct book library={            //此处容易漏掉等号
    "The Moon and Sixpence",
    "W.Somerset Maugham",    //各成员的初始化用逗号分隔
    2.97
};
```

使用点运算符访问结构中的成员，例如 library.value 就访问 value 成员，可以像使用任何 float 类型变量那样使用 library.value
C 为结构提供了指定初始化器，它使用点运算符，可以只初始化其中一个或部分成员，可以按照任意顺序初始化各个成员，对特定成员的最后一次赋值才是该成员实际获得的值

```
struct book gift={
    .value=2.97,
    .author="W.Somerset Maugham",
    18.90};  //赋给 value 的值是 18.90，因为它在结构声明中紧跟在 author 成员之后
```

声明结构数组：

```
struct book summit[maxbks];
```

以上代码把 summit 声明为一个内含 maxbks 个元素的数组，每一个元素都是一个 book 类型的结构体。summit[2].title[4]表示 summit 数组中的第 3 个元素（结构体）中 title 成员（书名部分）的第 5 个字符。
声明结构指针：

```
struct book *him;       //him 现在可以指向任何 book 类型的结构体
him=&blood;             //him 现在指向 blood
```

和数组不同的是，结构名并不是结构的地址，所以要在结构名前加&运算符
如果 him 是指向结构体 blood 的指针，则以下关系恒成立：

```
blood.title==(*him).title==him->title    //使用->运算符（间接成员运算符）
```

C 允许把一个结构赋值给另一个结构，即使结构的成员是数组，也能完成赋值。但是单独的数组之间不能直接赋值。
联合是可以在不同时刻保存不同类型和长度的成员的变量，使用和操作方式与结构相同，但它只能储存所有成员当中的一个，编译器负责跟踪成员的长度和对齐要求，并将分配足够的空间使它能够储存联合声明中占用最大字节的类型。关键字 struct 改为 union
32.统计输入中各个关键字出现的次数.C

```
#include<stdio.h>
#include<ctype.h>
      //包含函数 isspace 跳过空白字符，isalpha 识别字母，isalnum 识别字母和数字
#include<string.h>
#define MAXWORD 100                  /*关键字的最大长度*/
#define BUFSIZE 100                  /*缓冲区的长度*/
#define NKEYS 4                       /*关键字的个数*/
struct key{
/*声明一个结构数组，每个元素是一个对应关键字的结构体，其中存放指向关键字的指针和关键字的出现次数*/
    char *word;int count;
}keytab[NKEYS]={{"at",0},{"in",0},{"on",0},{"out",0}};
                                          //关键字必须按首字母顺序升序存储
int getword(char *,int);int binsearch(char *,struct key *,int);
char buf[BUFSIZE];
           //为了确定已经读入的输入是否足够，声明一个共享缓冲区（字符数组）
int bufp=0;                    //增加一个下标变量来记录缓冲区中当前字符的位置
/*getch 函数：从缓冲区里读取字符，缓冲区为空时则直接从输入中读取字符*/
int getch(void){return (bufp>0)?buf[--bufp]:getchar();}
/*ungetch 函数：把要压回输入的字符放到缓冲区里*/
void ungetch(int c)
{
    if (bufp>=BUFSIZE)
        printf("ungetch:too many characters\n");
    else
       buf[bufp++]=c;
}
/*binsearch 函数：折半查找，在结构体 tab[0]到 tab[n-1]中查找关键字*/
int binsearch(char *word,struct key tab[],int n)
{
                                  //tab 对应 main 中的 keytab,n 对应 NKEYS
    int cond,low=0,high=n-1,mid;             //数组元素必须以升序排列
    while(low<=high)
    {
    /*在这两种情况下，下一步都是将关键字与所选部分的中间元素进行比较，这个过程一直进行下去*/
        mid=(low+high)/2;            //首先将关键字与数组的中间元素进行比较
        if((cond=strcmp(word,tab[mid].word))<0)     //如果小于中间元素的值
           high=mid-1;                   //则继续在该数组的前半部分查找
        else if(cond>0)                                  //否则
           low=mid+1;                          //继续在该数组的后半部分查找
        else                                          //最后找到这个关键字时
           return mid;
      }
                    /*它一定和最后的中间元素相等，返回非负的数组下标 mid 值*/
     return -1;
}                                    //只要返回值非负就说明查到了
/*getword 函数：从输入中读取下一个单词，单词可以是以字母开头的字母和数字串，或一个非空白的字符。然后把这个单词复制到名字为该函数的第一个参数的数组中*/
int getword(char *word,int lim)
{
    int c;                                   //char 类型就是较小的整型
    int getch(void);void ungetch(int);          //在函数体内声明的函数原型
```

```
    char *w=word;  //声明一个指针指向和 word 相同的地址（关键字），从而实现复制
    while(isspace(c=getch()));
                            //用这样一条语句不断地读取输入字符并跳过空白字符
    if(c!=EOF)                          //上下两个 if 语句的测试条件相互对立
       *w++=c;                          //把读取的字符放在指针 w 指向的地址
    if(!isalpha(c))
    {
        *w='\0';
        return c;
    }                           //当读取的不是字母时，添加空字符
    for( ;--lim>0;w++)
        if(!isalnum(*w=getch()))
            {
                ungetch(*w);
                break;
            }
/*当读入的字符不属于字母数字的集合时，说明 getword 多读入了一个字符。随后调用
ungetch 将多读的一个字符放回到输入中，以便下一次调用使用*/
          *w='\0';
    return word[0];
}                       //函数返回值可能是单词的第一个字符，EOF,或字符本身
int main()
{
    int n;char word[MAXWORD];             //整型数 n 在 main 中有两种用途
    while(getword(word,MAXWORD)!=EOF)//反复调用 getword，每次读取一个单词
       if(isalpha(word[0]))
           if((n=binsearch(word,keytab,NKEYS))>=0)
                                        //每个单词将在 keytab 中进行查找
               keytab[n].count++;  //每查到一次就把相应关键字出现次数递增
    for(n=0;n<NKEYS;n++)
       if(keytab[n].count>0)            //没有出现的关键字不予打印
           printf("%4d %s\n",keytab[n].count,keytab[n].word);
    return 0; }
```

上面的例子中实现了函数与结构的交互，至少可以通过 3 种可能的方法传递结构：分别传递各个结构成员，传递整个结构，传递指向结构的指针。

```
/*makepoint 函数：通过 x,y 坐标构造一个点，带有 2 个整型参数并返回一个 point 类型的结构*/
struct point{int x;int y;};
struct point makepoint(int x,int y){
    struct point temp;
    temp.x=x;temp.y=y;
    return temp;}
```

C 提供了一个称为 typedef 的功能（**类型定义**），它能用来建立新的数据类型名，例如声明 typedef int length;将 length 定义为与 int 具有同等意义的名字。

typedef 声明并没有创建一个新类型，它只是为某个已存在的类型增加一个新的名称而已。typedef 声明也没有增加任何新的语义：通过这种方式声明的变量与通过普通声明方式声明的变量具有完全相同的属性。实际上 typedef 类似于#define 语句，但由于 typedef 是由编译器解释的，因此它的文本替换功能要强过预处理器的能力。例如：

```
typedef int (*pfi)(char *,char *);
pfi strcmp,numcmp;
```

该语句定义了类型 pfi 是“一个指向函数的指针，该函数具有 2 个 char *类型的参数，返回值类型为 int”。

typedef 可以使表达方式更简洁。使程序参数化，以提高程序的可移植性。

我们已经编写了一些 C 程序，现在来了解一下 C 语言

1972 年，贝尔实验室的丹尼斯·里奇(*Dennis Ritchie*)和肯·汤姆逊(*Ken Thompson*)在开发 UNIX 操作系统时发明了 C 语言。它具有可移植性，这意味着在一种系统中编写的 C 程序稍作修改或不作修改就能在其他系统运行，如果修改也只需简单更改主程序头文件中的少许项；程序员可以通过 C 来访问硬件、操控内存；C 代码紧凑简洁，结合指针和大量运算符可以写出让人极其费解又晦涩的代码。现在有一年一度的国际 C 语言混乱代码大赛

在 1990s，许多软件公司开始改用 C++来开发大型软件，C++在 C 的基础上嫁接了面向对象编程工具，C++几乎是 C 的超集，这意味着任何 C 程序差不多就是一个 C++程序，学习 C 也就相当于学习了许多 C++的知识。

C 是嵌入式系统编程的流行语言，也就是说目前很多现代化设备的微处理器都用 C 进行编程，C 在 Linux 开发中也极其重要

那么计算机是如何运行程序的呢？**中央处理器** CPU 承担绝大部分的运算工作，它是一种微处理器，**随机存取内存** RAM 是存储程序和文件的工作区。CPU 的工作很简单枯燥，就是从内存中重复获取并执行指令，CPU 有自己的小工作区——由若干个寄存器(register)组成，每个寄存器都可以储存一个数字。一个寄存器放下一条指令的内存地址，CPU 使用该地址来获取和更新下一条指令。在获取指令后 CPU 在另一个寄存器中储存该指令，并更新第 1 个寄存器储存下一条指令的地址。CPU 能够理解的指令(也就是计算机能听懂的语言)叫做**机器语言**，早期的编程都用机器语言。程序员用 0 和 1 数字组成的代码打在纸带或卡片上，1 打孔，0 不打孔，再将程序通过纸带机或卡片机输入计算机进行运算。

人们说话交流用的是**自然语言**，为了让人能够和计算机交流，而不是永无止境地打孔，科学家开发了**汇编语言**(Assembly)和**高级语言**。最早的高级编程语言是 Fortran，C 也是一种高级语言.

编译器(compiler)是把高级语言翻译成机器语言指令集的程序。

高级语言的源代码文本(*hello.c*) → 预处理器(preprocessor) → 编译器 → 汇编程序(assembler) → 目标代码(*hello.obj*) → 链接器(linker) → 可执行代码 executables (*hello.exe*)

库代码(比如 dynamical link library **动态链接库**.dll)　　启动代码

编程的 7 个步骤

定义程序的目标 → 设计程序 → 写代码 → 编译 → 运行(run) → 调试(debug) → 维护和修改代码

第 1 步和第 2 步是最重要也最容易被忽视的，这是设计**算法(algorithm)**的步骤，完全可以在大脑中完成。人们都说微软程序员一天只工作 1 小时，其实 TA 们用 7 个小时在大脑中设计思路，1 个小时用来写代码。

Debug 定律第 0 条：没有玄学，每个问题都有你学过的理论能解释的原因

Debug 定律第 1 条：问题的莫名其妙程度与问题的傻逼程度正相关

Debug 定律第 2 条：bug 的可重复性和解决难度负相关

伪代码是一种用简单的句子表示程序设计思路的方法，

现在回到 C 当中来，C 可以用来访问文件。

文件指针指向一个包含文件信息的结构，用户不需要关心这些信息，因为<stdio.h>中已经定义了一个包含这些信息的结构 FILE，注意 FILE 像 int 一样是一个类型名，而不是结构

标记。在读写一个文件之前，必须通过库函数 fopen 打开该文件，它以 2 个字符串作为参数并返回一个文件指针，第 1 个参数是文件名，第 2 个参数是访问模式，用于指定文件的使用方式。
允许的模式包括：读(“r”)写(“w”)追加(“a”)。某些系统还区分文本文件和二进制文件，对后者的访问需要在模式字符串中增加字符“b”。
如果打开一个不存在的文件用于写或追加，该文件将被创建。当以写方式打开一个已经存在的文件时，该文件原来的内容将被覆盖。但是，如果以追加方式打开一个文件，则该文件原来的内容将保留不变。读一个不存在的或无读取权限的文件将会出现错误，如果发生错误，fopen 将返回 NULL
流(stream)是与磁盘或其他外围设备关联的数据的源或目的地。打开一个流，将把该流与一个文件或设备连接起来，关闭流将断开这种连接。打开一个文件将返回一个文件指针，该指针指向的结构记录了控制该流的所有必要信息。在不引起歧义的情况下，我们不再区分“文件指针”和“流”
命令行操作是一项专业又酷炫的技能，常用的命令行软件有 *cmder/Cygwin,windows powershell* Windows 7 之后的版本自带 powershell。通常可以用 windows 键+R 开启运行，输入 cmd 再按 Enter 即可开启命令行界面(CLI)，电影中的黑客几乎不使用鼠标是因为他们懒得给自己开发的工具加上图形用户界面(GUI)，理论上可以只用键盘+CLI 操作电脑然而 Windows 的命令行非常难用，并且 99%的用户都不需要使用它
Linux 系统是在命令行上面做了个图形界面，Windows 系统是在图形界面顺便带了个命令行。Linux 所有的功能都可以用命令行实现，GUI 只是个 tool；Windows 所有的功能都可以用 GUI 实现，命令行只是个 tool。
C 提供了一些函数，头文件<math.h>中声明了数学函数
fabs(x) x 的绝对值 floor(x) 取整函数，表示不大于 x 的最大整型数 Sqrt(x) x 的平方根 pow(x,y)表示 x 的 y 次幂，x,y 都是 double 类型
注意 abs(x)定义在头文件<stdlib.h>中，表示整数的绝对值。<stdlib.h>包含了一些实用函数。而上面的数学函数的返回值通常是 double 类型。
<string.h>中定义了一些字符串操作函数：
将 t 指向的字符串连接到 s 指向的字符串的末尾 strcat(s,t)
将 t 指向的字符串的前 n 个字符连接到 s 指向的字符串的末尾 strncat(s,t,n)
按照首字母在字母表中排列顺序，比较 s 和 t 指向的字符串，s<t,s==t,s>t 分别返回负整数、0、正整数 strcmp(s,t)
将 t 指向的字符串复制到 s 指向的位置 strcpy(s,t)
<ctype.h>中定义了一些字符类别测试和转换函数，测试函数如果测得结果不符就会返回 0 测试是否是字母、大写字母、小写字母、数字、字母和数字的集合、空白字符。分别是 isalpha(c)isupper(c) islower(c) isdigit(c) isalnum(c) isspace(c)
返回字母 c 的大写形式 toupper(c) 返回 c 的小写形式 tolower(c)
for 循环中定义变量必须开启 c99 或 c11，具体方法是给 gcc 编译器加编译参数，使用命令：
gcc xxx.c-o xxx.exe-std=c99
如果不加编译参数，把代码修改一下即可：
33.打印九九乘法表.C

```
#include<stdio.h>
int main()
{
    int i,j;
    for(j=1,i=1;j<=i;j++)
    {
```

```
        printf("%d*%d=%d",i,j,i*j);
        j==i?(i++,j=0,printf("\n")):(printf(","));
                                        /*注意这里的分号、逗号、圆括号*/
        i==10?(j=i):(i=i);
    }
}
```

现在来写下第一个 C++程序：

1.Helloworld 程序.Cpp

```
#include<iostream>
int main()
{
    std::cout<<"hello world!\n";
    return 0;
}
```

只需要把源文件**扩展名**从.c 改成.cpp 即可在同样的 IDE 中运行 C++程序

在编程中设计一个好的算法可以事半功倍，例如辗转相除法(又名欧几里得算法，是已知最古老的算法)求最大公约数：

```
int gcd(int x,int y)
{
    return y?gcd(y,x%y):x;
}
```

从上面的例子可以看出，善于使用条件表达式可以写出精悍的短码

34.分解质因数.C

```
#include<stdio.h>
main(){
    long long ago,i;       /*i:循环标志，被初始化为最小的素数 2，是拿来除的*/
    printf("please enter a positive number:");
    scanf("%llu",&ago);
    printf("%llu=",ago);
    for(i=2;i<=ago;i++)
    {                                   //跳回这里来，把 i 的值加 1，重复执行
        while(ago!=i)
        {                               //如果 n 大于 2
            if(ago%i==0)
            {                           //但能被 2 整除
                printf("%llu*",i);                  //输出 2
                ago=ago/i;          //并用 n 除以 2 的商作为新的 n，重复执行
            }
            else
                break;
         }
     }                              //如果不能被 2 整除，跳出 while 循环
 printf("%llu\n",ago);
}                       //除到最后得到的 n 也是一个质因数，放在输出等式的最后
```

35.打印杨辉三角.C

```
#include<stdio.h>
                /*jiecheng 函数：递归求 x 的阶乘*/
float jiecheng(int x)
{
    return !x?1:x*jiecheng(x-1);
}
            /*zuheshu 函数：定义组合数，因为杨辉三角中的任何一个数都是组合数*/
float zuheshu(int i,int j)
```

```
{
    float k;
    k=jiecheng(j)/(jiecheng(i)*jiecheng(j-i));
    return k;
}
void main()
{
    int i=0,j,k,n;
    while(i<=0||i>16)
    {                   /*行数限制*/
        printf("please enter the lines you want:");
        scanf("%d",&i);
    }
    printf("%d lines Yang Hui triangle is this:\n",i);
    for(j=0;j<i;j++)
    {
     /*外层循环：每次循环打印一行，每一行先不断循环打印空格，再不断循环打印组合
数*/
        for(k=1;k<=(i-j);k++) /*内层循环 1：在打印组合数之前打印空格占位*/
            printf("    ");
        for(n=0;n<=j;n++) /*内层循环 2：不断地打印组合数，并且按格式化输出*/
            printf("%9.0f",zuheshu(n,j));
/*格式化输出数，这里的空格和内层循环 1 的空格不同，是格式化输出的结果*/
        printf("\n");
      }
      printf("\n\n");
}
```

当打印等腰三角形的时候，物理专业的同学总会联想到对称性。然而代码实际的运行过程中并没有在右边打印空格，它只是按顺序打印每一行。这是从物理思维转到计算机思维的一个典型例子

前面提到交换两个变量的值要用到临时变量（第 3 个变量），然而这里有一种方法，不使用第 3 者并且只用一行代码实现了交换：a^=b^=a^=b;

如果编译报错：undefined behavior(UB)，可以把代码展开：

```
a=a^b;
b=b^a;
a=a^b;
```

四舍五入法：加 0.5 然后取整

```
 float x;
 int y=(int)(x+0.5);
```

API(应用程序编程接口,Application Programming Interface)是一些预先定义的函数，目的是提供应用程序与开发人员基于某软件或硬件得以访问一组例程的能力，在 windows 中，它包含在动态链接库文件中。

36.求十亿内所有素数的和.C

```
#include<stdio.h>
#include<stdbool.h>
#include<stdlib.h>
#include<stdint.h>
#include<math.h>
#define MAX 1000000000
int main(void){
    bool *state=(bool*)malloc(sizeof(bool)*(MAX+1));
```

```
    int64_t sq=sqrt(MAX);
    for(int64_t i=2;i<=sq;++i)            //埃拉托色尼素数筛法
        if(!state[i])
            for(int64_t j=i*i;j<=MAX;j+=i)
                state[j]=true;
    int64_t sum=0;
    for(int64_t i=2;i<=MAX;++i)
        if(!state[i])
            sum += i;
    free(state);
    printf("sum=%lld\n",sum);
    return 0;
}
```

输出结果：

sum=24739512092254535

人类交流用的自然语言其实是一种信息的载体，交流本质上也就是一个通信的过程。

说话人(信息源)——→编码(组织语言)——→ 信息(信道，声音) ——→解码 ——→接收者

不同的语言之间的区别就是不同的编码与解码规则，传递的信息依然是相同的。语言会影响语言使用者的思维方式，是因为这种编码解码规则深深刻在了 TA 的潜意识中，影响了 TA 对信息的接收与处理方式。信息的冗余是信息安全的保障；语言的数据(语料)，尤其是双语或多种语言的对照语料对翻译至关重要。从象形文字到拼音文字是一个巨大的飞跃，因为人类在描述物体的方式上，从物体的外表进化到了抽象的概念，同时不自觉地采用了对信息的编码，而常用字短，生僻字长的特点完全符合信息论中的最短编码原理

在这里放一段 Java 程序，和 C 语言比较一下看看

```
import java.util.*;
public class Helloworld {
    public static double mysqrt(double c){      //牛顿迭代法计算平方根
        if(c < 0) return Double.NaN;
        double err = 1e-15;
        double t = c;
        while(Math.abs(t - c/t)>err*t)
            t = (c/t + t)/2.0;
        return t;
    }
    public static void main(String[] args){
        int a[] = {1,3,4,13,7,5};
        System.out.println(Arrays.toString(a));     //[1, 3, 4, 13, 7, 5]
        Arrays.sort(a);
        int location = Arrays.binarySearch(a, 4);
        System.out.println(location);               //2
        Arrays.fill(a, Integer.MAX_VALUE);
        System.out.println(Arrays.toString(a));
//[2147483647, 2147483647, 2147483647, 2147483647, 2147483647, 2147483647]
        int c[][]={{1,2,3},{4,5,6}};
        System.out.println(Arrays.deepToString(c));//[[1, 2, 3], [4, 5, 6]]
        System.out.println(mysqrt(3.0));     //1.7320508075688772
        List<List<Integer>> res = new LinkedList<>();
        res.add(Arrays.asList(1,2,4));
        System.out.println(res);     //[[1, 2, 4]]
    }
}
```

1956 年的达特茅斯夏季人工智能研究会共有 10 人参加，其中包括信息论的发明人香农以及

4 位图灵奖获得者。
让电脑完成翻译或语言识别需要**自然语言处理(NLP)** 早期科学家试图让计算机理解语法规则，画出句子的文法分析树，从而实现 nlp，然而 15 年来始终没有进展。最后人们转变了思维方式，开始为自然语言这种上下文相关的信息建立数学模型，即统计语言模型。
一个句子由一连串特定序列的词组成，现在我们想知道句子 S 在文本中出现的可能性，S 这个序列出现的概率是每一个词出现的概率的乘积，而其中一个词出现的概率取决于它前面的所有词(条件概率)。这样计算量太大，怎么办？我们引用**马尔可夫假设：**假设一个词出现的概率只和它前面那 1 个词有关，于是问题就简化了
而要计算这个条件概率，就要把历史上人类说过的写过的所有文字(语料库)全部拿来统计，根据大数定律，相对频率就相当于概率
符合马尔可夫假设的随机过程称为马尔可夫链，它随机地选择一个状态作为初始状态，然后按照某个规则随机选择后续状态，这样运行一段时间后就会产生一个状态序列，看到这个序列就能估计出 2 个状态之间的转移概率。隐含马尔可夫模型是马尔可夫链的扩展：任意时刻的状态 st 是不可见的，但是会在每个状态 st 下输出一个符号 ot，ot 只和 st 相关，这叫独立输出假设。隐含马尔可夫模型可用于解决通信的解码问题。
信息的作用在于消除不确定性，NLP 的大量问题就是寻找相关的信息。
1970 年代的 IBM 有点像 1990 年代的微软和 2000 年代的 Google，任由杰出科学家做自己感兴趣的研究。在这样的环境里，Frederek Jelinek 提出了统计语音识别的框架结构，在这之前，科学家把语音识别问题当作人工智能和模式匹配问题，而 Jelinek 把它当作通信问题，并用 2 个隐含马尔可夫模型把语音识别概括得清清楚楚。
香农信息熵的定义：$H = -(P1 * logp1 + p2 * logp2 + \cdots + pn * logpn)$
P1p2…pn 分别表示这 n 支球队夺冠的概率，H 就是信息“谁是冠军”的信息量，单位是比特。是一个 2 进制数，相当于你猜测谁是冠军的次数。计算机科学中 log 统一默认为以 2 为底(如果没有特殊说明)。
Google 有一道面试产品经理的考题：如何向你的奶奶解释搜索引擎。好的回答是拿图书馆的索引卡片做类比，每个网站就像图书馆里的一本书，我们不可能在图书馆的书架上一本一本地找，而是要通过搜索卡片找到它的位置，然后直接去书架上拿。通过建立索引而不是直接扫描所有的网站文本，我们使搜索引擎能够在 0.1 秒之内生成结果
图书馆的索引卡片当然无法进行复杂的逻辑运算，但是当信息检索进入计算机时代后，图书的索引不再是卡片而是基于数据库的。数据库的结构化查询语句(SQL)支持各种复杂的逻辑组合，但是背后的基本原理是基于布尔代数的。
离散数学是计算机科学的数学基础，它包括数理逻辑、集合论、图论、近世代数。数理逻辑基于布尔运算，布尔运算非常简单，全部内容就是三张真值表。
图论中讨论的图由一些节点和连接这些节点的弧组成，图论起源于欧拉七桥问题，图的遍历算法就是通过弧访问各个节点，主要有两种：**广度优先搜索** BFS(先要尽可能广地访问与每个节点直接连接的其他节点)和**深度优先搜索** DFS(一条路走到黑)。
互联网就是一张大图，每一个网页都是一个节点，那些超链接就是连接网页的弧，网络爬虫就是互联网的遍历算法，它可以爬取整个互联网上的特定信息。
Google 革命性的发明是 PageRank 网页排名算法，它使得搜索技术有了质的飞跃，它的发明者是拉里·佩奇和谢尔盖·布林。佩奇的想法是，如果一个网页被很多其他网页所链接，说明它受到普遍的承认和信赖，那么它的排名就高，这和民主表决一样。然而问题来了：计算搜索结果的网页排名过程中需要用到网页本身的排名，这是一个怪圈。布林破解了这个怪圈，他把这个问题转化成了二维矩阵乘法，并用迭代法解决。先假定所有网页排名的权重相同，根据初始值算出第一次迭代排名，不断进行下去，他们在理论上证明了不论初始值如何选取。

这种算法都能保证最终结果的估计值能够收敛到排名的真实值。实际问题中这个矩阵乘法的计算量非常大。Google 的科学家使用稀疏矩阵计算的技巧以及并行计算工具 MapReduce 让这个算法成功实现了完全自动化。

网页排名算法的高明之处在于它把互联网当成一个整体。而以前的信息检索大多把每一个网页当作独立的个体对待，大部分人当初只注意了网页内容和查询语句的相关性，忽略了网页之间的关系。

Google 的搜索质量第一技术负责人阿密特·辛格至今依然经常分析那些不好的搜索结果，但是很多做搜索的工程师都做不到这一点，他们总是指望靠一个算法、一个模型就能毕其功于一役，而这是不现实的。

Editor 操作问题 1　VScode 中文注释显示乱码的解决方法：点右下角的 UTF-8，上方正中出现”reopen with encoding”，点一下，输入 gbk 或 gb18030，选对了编码打开，就不会乱码了

通常，程序开发最重要的部分是找到程序中表示数据的好方法。然而，找出正确的数据表示不仅仅是选择一种数据类型，还要考虑必须进行哪些操作。也就是说必须确定如何储存数据，并且为数据类型定义有效的操作。

计算机科学的底层是数学，**数据**是**信息**的载体，是描述客观事物的数、字符以及所有能输入到计算机中并被计算机程序识别和处理的符号的集合。**数据元素**是数据的基本单位，通常作为一个整体来考虑和处理，它由**数据项**组成，数据项是组成数据元素的最小单位。**数据类型**是一个值的集合和定义在此集合上的一组操作的总称，比如整型、字符型。什么是类型？类型特指两类信息：属性和操作，int 类型的属性是它代表一个整数值，因此它共享整数的属性，而全体整数是一个值的集合，允许对 int 类型进行的操作是：加减乘除、改变符号、求模，当声明一个 int 类型的变量时，就表明了只能对该变量进行这些操作。

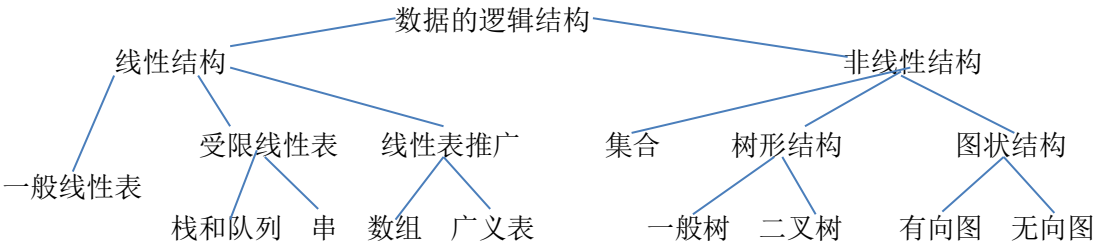
假设要定义一个新的数据类型，首先要提供储存数据的方法，然后要提供操作数据的方法

CS 领域已开发了一种定义新类型的好方法，用 3 个步骤完成从抽象到具体的过程。

1. 提供类型属性和相关操作的抽象描述，这些描述既不能依赖特定的实现，也不能依赖特定的编程语言。这种正式的抽象描述被称为**抽象数据类型**(ADT)
2. 开发一个实现 ADT 的编程接口。也就是说要指明如何储存数据和执行所需操作的函数这里包含了具体的实现细节，相当于“黑箱”的内部工作原理
3. 编写代码使用并实现接口。相当于使用这个黑箱

实际问题中数据元素不是孤立存在的，它们之前存在着关系，**数据结构**(data structure)是相互之间存在一种或多种特定关系的数据元素的集合，它包括逻辑结构、存储(物理)结构、数据的运算。**逻辑结构**独立于计算机之外。**存储结构**是数据结构在计算机中的表示(映像)

可以用 ADT 定义一个完整的数据结构



线性表(表)是具有相同数据类型的 n 个数据元素的有限序列，它的顺序存储称为**顺序表**，只允许在一端进行插入或删除操作的线性表叫做**栈**，只允许在一端进行插入而在另一端进行删除的线性表叫做**队列**，线性表的链式存储称为**单链表**，它通过一组任意的存储单元来存放表中元素，每个链表节点除了存放元素自身信息外，还存放一个指向后继节点的指针。为了

加快查找的速度，可以把关键码值映射到表中一个位置来访问记录，这个映射函数叫**散列函数**，存放记录的数组叫**散列表(哈希表)**，其原理是根据元素的关键字直接计算出该元素的存储地址。**循环队列**是用顺序表实现的一种数据结构，**树**是 n 个节点的有限集合，非空树有且只有一个特定的叫做**根**的节点，**二叉树**是一种特殊的树形结构，每个节点最多只有 2 颗子树，并且子树有左右之分，其次序不能随意颠倒。

算法是为求解一个问题需要遵循的、被清楚地指定的简单指令的集合，它具有 5 个特性：有穷性(总能够在执行有限步骤之后自动结束)，确定性(每一条指令必须有确切含义，相同的输入总给出相同的输出)，可行性，输入(有 n>=0 个输入)，输出(有 n>0 个输出)

好的算法具有：正确性，可读性，健壮性(输入非法数据时也能做出适当的反应和处理)，效率高(运行时间短)，运行需要的最大存储空间较少

数据结构主要研究组织大量数据的方法，**算法分析**是对算法运行时间的评估，数据结构与算法被认为是计算机科学中最有价值的部分，算法的核心是数学思想，**程序设计=数据结构+算法**

37.统计并按降序打印输入中的所有单词及其出现频率.C

```
/*模块 1：头文件区*/
#include<stdio.h>
#include<ctype.h>              /*提供 isalpha()原型*/
#include<stdlib.h>             /*提供 malloc()原型*/
#include<string.h>             /*提供 strdup()原型*/
/*模块 2：全局变量声明与初始化区*/
#define MAXWORD 100                /*对单个单词的长度设置限制*/
#define NDISTINCT 1000            /*对不同单词的最大数目设置限制*/
typedef struct tnode *treeptr;       /*指向节点的指针*/
typedef struct tnode{                  /*树的节点*/
    char *word;                    /*指向单词的指针*/
    int count;                        /*单词出现的次数*/
    treeptr left;                     /*左子节点*/
    treeptr right;             /*右子节点*/
} treenode;
treeptr list[NDISTINCT];             /*指针数组，保存指向节点的指针*/
int ntn=0;                  /*保存树节点的个数*/
/*模块 3：函数原型声明区*/
treeptr addtree(treeptr,char *);
int getword(char *,int);
void sortlist(void);
void treestore(treeptr);
treeptr talloc(void);
/*模块 4：主程序区*/
int main()
{
    treeptr root;
    char word[MAXWORD];
    int i;
    root=NULL;
    while(getword(word,MAXWORD)!=EOF)
        if(isalpha(word[0]))
            root=addtree(root,word);          /*循环生成二叉树*/
    treestore(root);              /*从根节点开始遍历二叉树*/
    sortlist();
    for(i=0;i<ntn;i++)
```

```
        printf("%2d:%20s\n",list[i]->count,list[i]->word);
    return 0;
}
/*模块 5：talloc 函数：创建一个节点 tnode*/
treeptr talloc(void)
{
    return (treeptr)malloc(sizeof(treenode));
}
/*模块 6：addtree 函数：在 p 的位置或 p 的下方增加一个 w 节点*/
treeptr addtree(treeptr p,char *w)
{
    int cond;
    if(p==NULL)
    {                /*该单词是一个新单词*/
        p=talloc();                /*创建一个新节点*/
        p->word=strdup(w);        /*stdrup 函数复制 w 到某个安全位置*/
        p->count=1;
        p->left=p->right=NULL;
    }
    else if((cond=strcmp(w,p->word))==0)
        p->count++;                /*新单词与节点中的单词匹配*/
    else if(cond<0)            /*如果小于该节点中的单词，则进入左子树*/
        p->left=addtree(p->left,w);
    else                        /*如果大于该节点中的单词，则进入右子树*/
        p->right=addtree(p->right,w);
    return p;
}
/*模块 7：treestore 函数：把所有指向节点的指针保存到数组 list 中*/
void treestore(treeptr p)
{
    if(p!=NULL)
    {
        treestore(p->left);
        if(ntn<NDISTINCT)
            list[ntn++]=p;
        treestore(p->right);
    }
}
/*模块 8：改进之后的数组 list 的希尔排序算法*/
void sortlist()
{
    int gap,i,j;
    treeptr temp;
    for(gap=ntn/2;gap>0;gap/=2)
        for(i=gap;i<ntn;i++)
           for(j=i-gap;j>=0;j-=gap)
           {
              if((list[j]->count)>=(list[j+gap]->count))
                  break;
              temp=list[j];
              list[j]=list[j+gap];
              list[j+gap]=temp;
            }
}
/*模块 9：getword 函数*/
```

```
/*模块 9.1：变量声明区*/
#define BUFSIZE 100
char buf[BUFSIZE];
int bufp=0;
/*模块 9.2：getword:get next word or character from input*/
int getword(char *word,int lim){
    int c,getch(void);
    void ungetch(int);
    char *w=word;
    while(isspace(c=getch()));
    if(c!=EOF)
       *w++=c;
    if(!isalpha(c)){*w='\0';return c;}
    for( ;--lim>0;w++)
       if(!isalnum(*w=getch())){ungetch(*w);break;}
    *w='\0';
    return word[0];}
/*模块 9.3：getch 和 ungetch 函数*/
int getch(void){return (bufp>0)?buf[--bufp]:getchar();}
void ungetch(int c){
    if (bufp>=BUFSIZE){printf("ungetch:too many characters\n");}
    else
       buf[bufp++]=c;}
```

对于大量的输入数据，表的线性访问时间太慢，在这个例子中我们用到了**二叉查找树**

每个不同的单词在树中都是一个节点，每个节点包含：一个指向该单词内容的指针、一个统计出现次数的计数值、一个指向左子树的指针、一个指向右子树的指针。

在读取输入中任意单词的同时，就将它放置到正确的位置，这个步骤由函数 addtree 实现这个函数是递归的，主函数 main 以参数的方式传递给该函数的一个单词将作为树的最顶层(根)。在每一步中，新单词与节点中存储的单词进行比较，随后通过递归调用 addtree 而转向左子树或右子树，该单词最终将与树中的某节点相匹配。这样一来，任何节点的左子树只包含按字典顺序小于该节点中单词的那些单词，右子树则反之。

38.逆波兰计算器.C

```
#include<stdio.h>
#include<stdlib.h>                  /*提供 atof()原型*/
#include<ctype.h>
#define MAXOP 100                       /*操作数或运算符的最大长度*/
#define NUMBER '0'                 /*标识找到一个数*/
int getop(char []);
void push(double);
double pop(void);
int main()
{
    int type;
    double op2;
    char s[MAXOP];
    while((type=getop(s))!=EOF)
    {                              /*下一个运算符或操作数不是文件结束指示符*/
        switch (type)
        {
            case NUMBER:                            /*if(是数)*/
                 push(atof(s));                    /*将该数压入到栈中*/
                 break;    /*atof 函数：把字符串 s 转换为相应的双精度浮点数*/
            case '+':                          /*else if(是运算符)*/
```

```
                push(pop()+pop());
                        /*弹出所需数目的操作数，执行运算，将结果压入到栈中*/
                break;
            case '*':
                push(pop()*pop());
                break;
            case '-':       /*减法和除法运算不满足交换律，必须区分左右操作数*/
                op2=pop(); /*为保证正确次序，把第一个值弹出到一个临时变量中*/
                push(pop()-op2);
                break;
            case '/':
                op2=pop();
                if(op2!=0.0)                    /*分母不能为 0*/
                   push(pop()/op2);
                else
                   printf("error:zero divisor\n"); /*对非法输入适当处理*/
                break;
            case '\n':                              /*else if(是换行符)*/
                printf("\t%.8g\n",pop());           /*弹出并打印栈顶的值*/
                break;
            default:                                            /*else*/
                printf("error:unknown command %s\n",s);        /*出错*/
                break; }}
    return 0;
}
/*模块 1：栈的操作*/
#define MAXVAL 100                /*栈 val 的最大深度*/
int sp=0;                   /*下一个空闲栈位置*/
double val[MAXVAL];           /*值栈*/
/*模块 1.1：push 函数：把 f 压入到栈中*/
void push(double f)
{
    if(sp<MAXVAL)
       val[sp++]=f;
    else
       printf("error:stack full,can't push %g\n",f);
}
/*模块 1.2：pop 函数：弹出并返回栈顶的值*/
double pop(void)
{
    if(sp>0)
       return val[--sp];
    else
    {
       printf("error:stack empty\n");
       return 0.0;
    }
}
/*模块 2：getop 函数：获取下一个运算符或数值操作数*/
int getop(char s[])
{
    int i,c,getch(void);
    void ungetch(int);
    while((s[0]=c=getch())==' '||c=='\t');      /*跳过空格与制表符*/
    s[1]='\0';
```

```
    if(!isdigit(c)&&c!='.')            /*如果下一个字符不是数字或小数点*/
        return c;           /*则返回*/
    i=0;
    if(isdigit(c))              /*否则，收集整数部分*/
        while(isdigit(s[++i]=c=getch()));     /*把这些数字字符串收集起来*/
    if(c=='.')               /*其中可能包含小数点，收集小数部分*/
        while(isdigit(s[++i]=c=getch()));
    s[i]='\0';
    if(c!=EOF)
        ungetch(c);
    return NUMBER;            /*并返回 NUMBER，以标识数已经收集起来了*/
}
/*模块 3：getch 和 ungetch 函数*/
#define BUFSIZE 100
char buf[BUFSIZE];
int bufp=0;
int getch(void){return (bufp>0)?buf[--bufp]:getchar();}
void ungetch(int c){
    if (bufp>=BUFSIZE){printf("ungetch:too many characters\n");}
    else
       buf[bufp++]=c;}
```

计算机喜欢一种用来代替普通中缀表示法的逆波兰表示法，在逆波兰表示法中，所有运算符都跟在操作数的后面，比如(1-2)*(4+5)用逆波兰表示为 1 2-4 5+*

逆波兰表示法中不需要圆括号，只要知道每个运算符需要几个操作数就不会引起歧义

程序实现很简单，每个操作数都被依次压入到栈中(把栈看作一口井)，当一个运算符到达时，从栈中弹出 2 个操作数并把该运算符作用于它们上面，再把运算结果再压入到栈中，这样不断进行下去就形成了一个循环，到达输入行末尾时就把最终运算结果弹出并打印。从左到右读取输入行，当见到一个数就把它堆到栈上(把栈想象成一叠碗，你只能取放最上面的碗)，这样是按输入顺序堆进来的，见到一个运算符时，就把上面的 2 个数弹出来，然后把运算符作用到它们上面再把结果再堆上去，如此循环。

单元测试就是对代码的基本逻辑单元进行测试，一般是函数、类，最开始学编程时是在 main 函数里面注释掉其他代码自己写一段测试一下功能，其实这就是简单的单元测试，但是这样不方便，目前的 IDE 都可以创建单元测试的工程，把单元测试的工程和你的项目工程关联起来，然后 unit test 工程里每一个函数就对应一个 unit test，运行的时候每个函数都会被执行一遍。当然这属于 IDE 的高级功能，这里不做介绍。

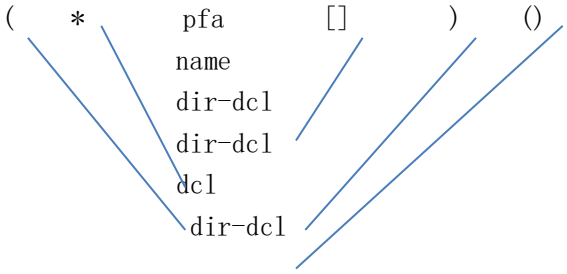
我们设计一个分析 C 语言指针声明的程序，声明符 dcl 就是前面可能带有多个*的 dir-dcl，dir-dcl 可以是 name、由一对圆括号括起来的 dcl、后面跟着一对圆括号的 dir-dcl、后面跟着用方括号括起来的表示可选长度的 dir-dcl。考虑这样的声明：

(*pfa[])()，pfa 将被识别为一个 name，即 dir-dcl，于是 pfa[]也是一个 dir-dcl，接着*pfa[]被识别为 dcl，因此(*pfa[])是一个 dir-dcl，然后(*pfa[])()被识别为一个 dir-dcl，因此也是一个 dcl

语法分析树如下：

dcl: 前面带有可选的*的 dir-dcl

dir-dcl: name
          (dcl)
          dir-dcl()
          dir-dcl[可选的长度]



dir-dcl

dcl

该程序的核心是两个函数：dcl 和 dirdcl，它们根据声明符的语法对声明进行分析。因为语法是递归定义的，所以在识别一个声明的组成部分时，这两个函数是相互递归调用的，我们称该程序是一个递归下降语法分析程序

39.分析 C 语言的指针声明(cdecl 程序的不完整版本).C

```
#include<stdio.h>
#include<string.h>
#include<ctype.h>
#define MAXTOKEN 100
enum{NAME,PARENS,BRACKETS};
void dcl(void);void dirdcl(void);
int gettoken(void);
int tokentype;                              /*最后一个记号的类型*/
char token[MAXTOKEN];                             /*最后一个记号字符串*/
char name[MAXTOKEN];                          /*标识符名*/
char datatype[MAXTOKEN];                  /*数据类型为 char,int 等*/
char out[1000];                           /*输出串*/
int main()
{
    while(gettoken()!=EOF)
    {
        strcpy(datatype,token);
        out[0]='\0';
        dcl();
        if(tokentype!='\n')
            printf("syntax error\n");
        printf("%s: %s %s\n",name,out,datatype);}
    return 0;
}
int gettoken(void)
{
    int c,getch(void);void ungetch(int);
    char *p=token;
    while((c=getch())==' '||c=='\t');           /*跳过空格和制表符*/
    if(c=='(')
    {
        if((c=getch())==')')
        {
            strcpy(token,"()");
            return tokentype=PARENS;
        }
        else
        {
            ungetch(c);
            return tokentype='(';
        }
    }
    else if(c=='[')
    {
        for(*p++=c;(*p++=getch())!=']'; );
        *p='\0';
        return tokentype=BRACKETS;
    }
```

```
    else if(isalpha(c))
    {
        for(*p++=c;isalnum(c=getch()); )
            *p++=c;
        *p='\0';
        ungetch(c);
        return tokentype=NAME;
    }
    else
        return tokentype=c;
}
void dcl(void)
{
    int ns;
    for(ns=0;gettoken()=='*'; )                    /*统计字符*的个数*/
        ns++;
    dirdcl();
    while(ns-->0)
        strcat(out," pointer to");
}
void dirdcl(void)
{
    int type;
    if(tokentype=='(')
    {                           /*形式为(dcl)*/
        dcl();
        if(tokentype!=')')
           printf("error:missing )\n");
    }
    else if(tokentype==NAME)                        /*变量名*/
        strcpy(name,token);
    else
        printf("error:expected name or (dcl)\n");
    while((type=gettoken())==PARENS||type==BRACKETS)
        if(type==PARENS)
            strcat(out," function returning");
        else
        {
            strcat(out," array");
            strcat(out,token);
            strcat(out," of");
        }
}
#define BUFSIZE 100
char buf[BUFSIZE];
int bufp=0;
int getch(void){return (bufp>0)?buf[--bufp]:getchar();}
void ungetch(int c){
    if (bufp<BUFSIZE)
       buf[bufp++]=c;}
```

例如输入 char (*(*x[3])())[5]　　分析结果为

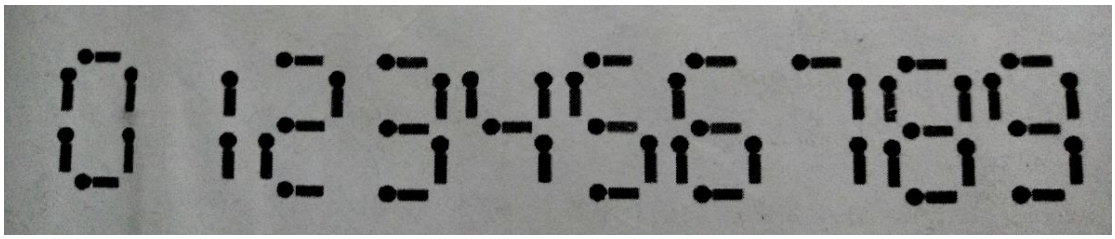
x: array[3] of pointer to function returning pointer to array[5] of char

40.直角坐标转换为极坐标.C

```
#include<stdio.h>
#include<math.h>
```

```
#define RAD_TO_DEG (180/(4*atan(1)))
typedef struct polar_v
{
    double magnitude;
    double angle;
} Polar_v;
typedef struct rect_v
{
    int x; /*如果设置成 double 型但是输入不带小数点的数据，运行可能得到 0 结果*/
    int y;
} Rect_v;
Polar_v rect_to_polar(Rect_v);
int main(void)
{
    Rect_v input;
    Polar_v result;
    puts("enter x and y coordinates;enter an alpha to quit:");
    while(scanf("%d %d",&input.x,&input.y)==2)
    {
        result=rect_to_polar(input);
 printf("magnitude=%0.2f,angle=%0.2f\n",result.magnitude,result.angle);
    }
    puts("Bye.");
    return 0;
}
Polar_v rect_to_polar(Rect_v rv)
{
    Polar_v pv;
    pv.magnitude=sqrt(rv.x*rv.x+rv.y*rv.y);
    if(pv.magnitude==0)
       pv.angle=0.0;
    else
       pv.angle=RAD_TO_DEG*atan2(rv.y,rv.x);
    return pv;}
```

现在来玩个游戏，我有 m 根火柴棍，那么我可以拼出多少个形状是 A+B=C 的等式呢？
注意：加号和等号各自需要两根火柴，数字 0~9 的拼法如下：



解决这个问题可以使用**枚举算法(穷举法,Brute Force)**，是一种相当暴力的破解办法，它的核心思想是：有序地去尝试每一种可能
分别枚举 A,B,C，需要确定它们的范围，我们假设已知条件是 m<=24，除去加号和等号还有 20 根火柴，在 0~9 这些数字中，数字 1 用到的火柴数最少，只需要 2 根。20 根火柴棍最多能组成 10 个 1，因此我们可以假定 A,B,C 中的任何一个数都不能超过 11111
A 所使用的火柴数+B 所使用的火柴数+C 所使用的火柴数=m-4，满足这个等式就可以找出一组解。枚举算法可能会有复杂度太高的情况出现，为了降低算法运行的复杂度，我们只枚举 A 和 B，而 C 可以通过 A+B 计算出来

41.火柴棍等式.C

```
#include<stdio.h>
```

```
int fun(int x)
{                       /*计算一个数 x 所用的火柴总数*/
    int num=0;                          /*用来计数的变量一定要初始化为 0*/
    int firenumber[10]={6,2,5,5,4,5,6,3,7,6};
                            /*用一个数组来记录 0~9 每个数字各需要的火柴数*/
    while(x/10!=0)
    {                       /*如果 x 至少有两位*/
        num+=firenumber[x%10]; /*获得 x 的末位数所需的火柴数并累加到结果中*/
        x=x/10;
    }                           /*去掉 x 的末位数，循环继续*/
    num+=firenumber[x];                     /*循环之后，现在 x 是一个个位数了*/
    return num;}                            /*返回结果*/
int main()
{
    int a,b,c,m,sum=0;                      /*sum 是用来计数的，也初始化为 0*/
    scanf("%d",&m);
    for(a=0;a<=11111;a++)
    {                                   /*用 2 个嵌套的 for 循环来枚举这 2 个数*/
        for(b=0;b<=11111;b++)
        {
            c=a+b;
            if(fun(a)+fun(b)+fun(c)==m-4)
            {                                       /*循环内部的判断执行语句*/
                printf("%d+%d=%d\n",a,b,c);
                sum++;
            }
        }
    }   /*sum 用来记等式数目，所以和等式的打印语句放一起*/
    printf("一共可以拼出%d 个不同的等式",sum);
    getchar();getchar();
    return 0;}
```

C 语言中有一个用于排序的标准库函数 qsort()，它的原型是

void qsort(void *base,size_t nmemb,size_t size,int (*compar)(const void*,const void*))

第 1 个参数是指向待排序数组首元素的指针，由于 C 允许把指向任何类型的指针强制转换成指向 void 的指针，所以 qsort()的第 1 个实际参数可以引用任何类型的数组。

第 2 个参数是待排序项的数量，即数组元素的数量。第 3 个参数是数组中每个元素占用的空间大小。

第 4 个参数是一个指向函数的指针，该函数返回 int 类型的值且接受两个指向 const void 的指针作为参数，这是一个比较函数，由你自己来编写，记住：函数名作为参数时就是指向该函数的指针。

假如我们要比较两个 double 类型数据的大小(它们是一个 double 类型数组中的两项)，如何编写这个比较函数？qsort()把两个待比较元素的地址传递给比较函数

int mycomp(const void *p1,const void *p2);把待比较的两个 double 类型值的地址赋给 p1 和 p2。这里存在一个问题，为了比较指针所指向的值，必须解引用指针，因为值是 double 类型，所以要把指针解引用为 double 类型的值，然而 qsort()要求指针指向 void，要解决这个问题，必须在比较函数内部声明两个类型正确的指针，并初始化它们分别指向作为参数传入的值

42.按每一行的长度排序输出文本.C

```
#include<stdio.h>
```

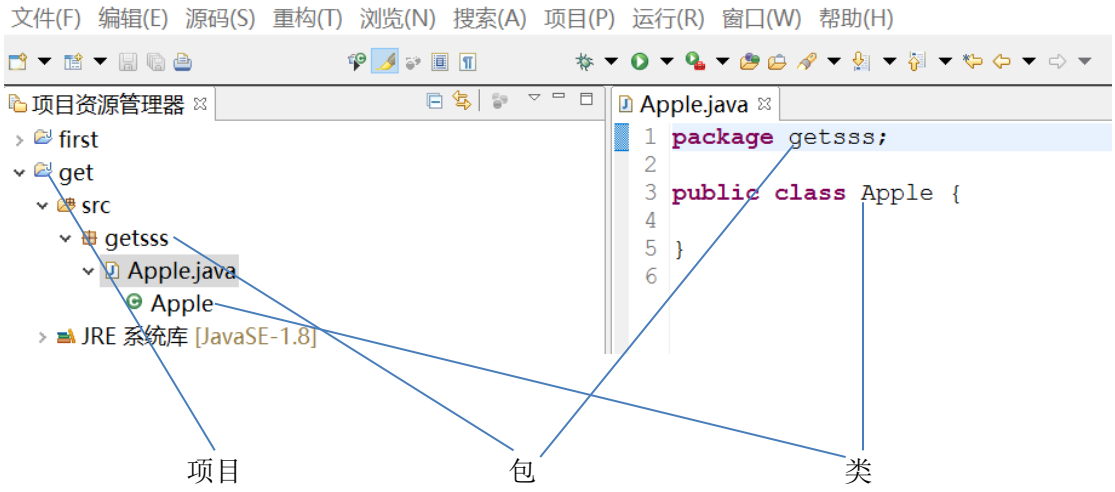
```
#include<string.h>
#include<stdlib.h>
#define MAXLINE 5000                            /*进行排序的最大文本行数*/
char *lineptr[MAXLINE];                         /*指向文本行的指针数组*/
int readlines(char *lineptr[],int nlines);
void writelines(char *lineptr[],int nlines);
#define MAXLEN 1000                         /*每个输入文本行的最大长度*/
int getline(char *,int);
char *alloc(int);
/*readlines 函数：读取输入行*/
int readlines(char *lineptr[],int maxlines)
{
    int len,nlines;                  /*len:每一行的长度，nlines:行数*/
    char *p,line[MAXLEN];  /*p:指向 char 类型的指针，line:存储文本行的数组*/
    nlines=0;                                       /*初始化行数*/
    while((len=getline(line,MAXLEN))>0)
        if(nlines>=maxlines||(p=alloc(len))==NULL)
                       /*把指针 p 指向长度为 len 的字符串，避免 p 成为野指针*/
            return -1;                     /*maxlines:最大行数*/
        else
        {
            line[len-1]='\0';                     /*删除换行符*/
            strcpy(p,line);                   /*把指针 p 指向文本行*/
            lineptr[nlines++]=p;
        }       /*把指针 p 存放到指针数组中*/
        return nlines;
}               /*返回读取的输入行数目*/
/*writelines 函数：写输出行*/
void writelines(char *lineptr[],int nlines)
{
    int i;
    for(i=0;i<nlines;i++)
        printf("%s\n",lineptr[i]);
}
int getline(char *line,int max)
{
    if(fgets(line,max,stdin)==NULL)
        return 0;
    else
        return strlen(line);
}
/*alloc 函数：返回指向 n 个字符的指针*/
#define ALLOCSIZE 10000              /*可用空间大小*/
static char allocbuf[ALLOCSIZE];          /*alloc 使用的存储区*/
static char *allocp=allocbuf;         /*指向下一个空闲的存储位置*/
char *alloc(int n)
{
    if(allocbuf+ALLOCSIZE-allocp>=n)
    {           /*有足够的空闲空间*/
        allocp+=n;
        return allocp-n;          /*分配前的指针 p*/
    }
    else                    /*空闲空间不够*/
        return 0;
```

```
}
int mycomp(const void *p1,const void *p2)
{
    const int *a1=(const int *) p1;          /*解引用指针*/
    const int *a2=(const int *) p2;
    if(strlen(*a1)<strlen(*a2))  /*[Warning]  passing  argument  1  of
'strlen' makes pointer from integer without a cast*/
       return -1;
    else if(strlen(*a1)==strlen(*a2))     /*[Note] expected 'const char *'
but argument is of type 'int'*/
       return 0;
    else
       return 1;
};
main(){
    int nlines;
    if((nlines=readlines(lineptr,MAXLINE))>=0)
    {
        qsort(lineptr,nlines,sizeof(char *),mycomp);
        writelines(lineptr,nlines);
        return 0;
    }
    else
    {
        printf("error:input too big to sort\n");
        return 1;}}
```

这个程序出现了 warning，有经验的人会明白：今天的 warning 就是明天的 error，容忍警告的程序员是不能容忍的，世界上 99%的项目都是瞎搞的，正经的项目都是 treat all warning as error 的

现在我们来用 eclipse 写下第一个 Java 程序，首先打开 eclipse，按以下顺序操作：

文件——新建——项目——选择向导，输入过滤器文本(不要动)，选 Java 项目——下一步——取个项目名(使用缺省位置，为源文件和类文件创建单独的文件夹)——完成——要打开透视图吗？如果否，可以直接在左边看到新建的项目，如果是，则还要打开透视图并选择”资源”才能看到新建的项目——点击项目，看到 src——右键点击——新建——类(如果直接新建类，系统会自动给包取一个和项目相同的名字，你也可以在这里点击新建包然后自己取个包名)——取个类名——完成——开始编程



这就是完成之后的效果

1.Hello world 程序.java

```
package yes;

public class Helloworld {
      public static void main(String args[]){
          System.out.println("hello world");
      }
}
```

在后面的 java 示例程序中，我将不会再在第一行包含包的名称

2.第一个图形用户接口(GUI).java

```
import javax.swing.*;          //别忘了 import 进这个包
public class Apple {
        public static void main(String args[]){
            JFrame frame=new JFrame();          //创建 frame 和 button
            JButton button=new JButton("click me!");
            frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                                                //在 windows 关闭时结束程序
            frame.getContentPane().add(button);
                                                //把 button 加到 frame 的 pane 上
            frame.setSize(300,300);             //设定 frame 的大小
            frame.setVisible(true);}}           //最后把 frame 显示出来!
```

点击运行之后会生成上方的那样一个窗口，窗口里面有一个大按钮，但是按钮并没有功能

Java 的工作方式：编写源代码 party.java——用编译器运行源代码——生成 party.class 文件(由字节码组成)——启动 java 虚拟机(JVM)来运行 party.class 文件，JVM 会将字节码转换为平台能够理解的方式来运行

Java 的程序结构：

```
 Source file(~~~.java)  源文件带有类的定义，通常只包含一个类
    Class file              类中带有一个或多个方法，
     Method 1             方法必须在类的内部声明
     Statement….. 方法代码是由一组语句组成，java 中的方法类似于 C 中的函数
     Method 2
     Statement…..
```

Java 中最常见的方法是 main 方法 public static void main(String[] args){…..}

public 表示公开给其他类存取，void 表示没有返回值，main 是方法名，String[] args 表示必须要传递 String 数组给此方法做参数，并命名为 args

类并非必须包含 main 方法，但是只有包含 main 方法的源文件才能够被执行，main()的两种用途是：测试真正的类，启动你的 Java 应用程序

编程中如果需要多个类，你可以在包里面新建几个类，这样会生成几个源文件，每个源文件包含一个类，只要把类声明为 public，它们之间就可以相互存取，从而可以作为一个整体来运行

面向对象(object oriented,OO) 是一门哲学，它通过对语言建模来适应问题，而不是对问题建模以适应语言，**面向过程(procedure oriented)**就是分析出解决问题需要的步骤，然后设计算法一步一步实现，

你的程序要完成一个任务，相当于讲一个故事。面向过程是编年体，面向对象是纪传体。对于复杂的程序(宏大的叙事)，面向对象(纪传体)是更为合理的讲述方法

面向对象可以帮助程序员用更自然的方法设计，加入新功能时不需要改动之前已经测试好的代码，它将数据和操作数据的方法放在同一个类当中，而类可以重复运用在别的应用程序中

类不是对象，却是用来创建它们的模型，类是对象的蓝图。
对象本身已知的事物被称为实例变量，它代表对象的状态；对象可以执行的动作被称为方法，它代表对象的行为。

```
public class dog{         //编写一个类
    int size;           //实例变量
    String breed;
    String name;
    void bark(){          //方法
        System.out.println("Ruff! Ruff!");
    }//方法结束
}//类结束
public class dogtestdrive{            //编写一个测试用的类
    public static void main(String[] args){
        dog d=new dog();           //新建一个 dog 对象
        d.size=40;            //用圆点运算符存取该对象的变量
        d.name="kevin";
        d.bark();}}            //调用它的方法
```

所有的 Java 程序代码都必须呆在类中。

3.无聊的猜数字游戏.java

```
public class Guessgame {
    player p1;         //用 3 个实例变量分别表示 3 个 player 对象
    player p2;
    player p3;
    public void startgame(){
        p1=new player();           //创建出 player 对象
        p2=new player();
        p3=new player();
        int guessp1=0;                //声明 3 个变量来保存猜测的数字
        int guessp2=0;
        int guessp3=0;
        boolean p1isright=false;      //声明 3 个变量来保存是否猜中
        boolean p2isright=false;
        boolean p3isright=false;
        int targetnumber=(int)(Math.random()*10);       //产生谜底数字
        System.out.println("I'm thinking of a number between 0 and 9...");
        while(true){
           System.out.println("Number to guess is "+targetnumber);
           p1.guess();                    //调用 player 的 guess 方法
           p2.guess();
           p3.guess();
           guessp1=p1.number;     //取得每个 player 所猜测的数字并将它们列出
           System.out.println("player one guessed "+guessp1);
           guessp2=p2.number;
           System.out.println("player two guessed "+guessp2);
           guessp3=p3.number;
           System.out.println("player three guessed "+guessp3);
           if(guessp1==targetnumber){p1isright=true;}
                          //检查是否猜中，如果猜中就去设定是否猜中的布尔变量
           if(guessp2==targetnumber){p2isright=true;}
           if(guessp3==targetnumber){p3isright=true;}
           if(p1isright||p2isright||p3isright){     //如果至少有一个猜中
               System.out.println("we have a winner!");
               System.out.println("player one got it right? "+p1isright);
```

```
                System.out.println("player two got it right? "+p2isright);
                System.out.println("player three got it right? "+p3isright);
                System.out.println("game is over.");
                break;      //游戏结束，终止循环
            }else{          //如果都没猜中，就重复循环继续猜下去
                System.out.println("players will have to try again.");
    }//if/else 结束
    }//循环结束
}//方法结束
}//类结束
public class player {
    int number=0;                //要被猜的数字
    public void guess(){
          number=(int)(Math.random()*10);
//Math 是一个标准的类，调用它的 random 方法生成 0 到 1 之间的浮点型随机数，乘以 10 并经过强制类型转换
          System.out.println("I'm guessing "+number);}}
public class GameLauncher {
    public static void main(String[] args){
        Guessgame game=new Guessgame();
        game.startgame();}}
```

下面是运行结果，每次都会不同的：

```
I'm thinking of a number between 0 and 9...
Number to guess is 3
I'm guessing 0
I'm guessing 1
I'm guessing 1
player one guessed 0
player two guessed 1
player three guessed 1
players will have to try again.
Number to guess is 3
I'm guessing 1
I'm guessing 3
I'm guessing 9
player one guessed 1
player two guessed 3
player three guessed 9
we have a winner!
player one got it right? false
player two got it right? true
player three got it right? false
game is over.
```

接下来我们继续回到 C/C++的世界中来，本书中的大多数算法和数据结构都将用 C/C++来描述

C++程序由一个或多个被称为函数的模块所组成，来看一个例子

2.计算阶乘.Cpp

```
#include<iostream>
```

```
const int arsize=16;
int main()
{
    using namespace std;
    long long factorials[arsize];
    factorials[1]=factorials[0]=1LL;
    for(int i=2;i<arsize;i++)
        factorials[i]=i*factorials[i-1];
    for(int i=0;i<arsize;i++)
        cout<<i<<"!="<<factorials[i]<<endl;
    return 0;
}
```

像 iostream 这样的就是头文件，有些 C 头文件被转换成 C++头文件，被重新命名去掉了扩展名.h 并加上前缀 c 表示来自 C 语言。例如 C++版本的 math.h 是 cmath

using namespace std;

这叫做 using 编译指令，namespace 是名称空间的意思，名称空间有助于组织程序中使用的标识符，避免名称冲突。由于标准库是使用新的头文件组织实现的，它将名称放在 std 名称空间中，使用编译指令 using 来使 std 名称空间中的名称可用

如果把 using 指令放在函数内部，那么名称就只在这个函数内可用，如果放在程序开头，那么程序中所有函数都可使用

main 不是关键字，但它是一个必不可少的函数的名称

类型是程序设计最基本的概念之一，它不仅定义了数据元素的内容，还定义了这类数据上可以进行的操作。

C++并未定义任何输入输出(IO)语句，取而代之包含了一个全面的标准库来提供 IO 机制，其中 iostream 库包含两个基础类型 istream 和 ostream，分别表示输入流和输出流，术语“流”想要表达的是，随着时间的推移，字符是顺序生成或消耗的

标准库定义了 4 个 IO 对象，cin(标准输入，发音：see-in)，cout(标准输出，see-out)，cerr(标准出错)，clog。其中只有 cin 是 istream 类型的对象

3.求输入整数的和.Cpp

```
#include<iostream>
int main()
{
    int sum=0,value=0;
    while(std::cin>>value)
        sum+=value;
    std::cout<<" sum is: "<< sum <<std::endl;
    return 0;
}
```

<<和>>分别表示输出运算符和输入运算符，::表示作用域运算符，通过使用作用域运算符来指出我们想使用定义在命名空间 std 中的名字 cout，<<和>>都接受两个运算对象，并返回其左侧运算对象作为其计算结果，这样可以把待输出的字符串字面值常量拼接起来

endl 是一个被称为操纵符的特殊值，它的效果是换行

在上面的程序中，使用 value 保存用户输入的每个数，数据的读取操作是在 while 的循环条件中完成的，循环条件的求值就是执行表达式 std::cin>>value

此表达式从标准输入读取下一个数并保存在 value 中，>>返回其左侧运算对象，因此循环条件实际上检测的是 std::cin

当我们使用一个 istream 对象作为条件时，其效果是检测流的状态，如果流是有效的，即流未遇到错误，那么检测成功。当遇到文件结束符或无效输入(比如不是整数)，istream 对象的状态会变为无效，处于无效状态的 istream 对象会使条件变为假。因此 while 循环

会一直执行直至遇到文件结束符
标准库类型 string 表示可变长度的字符序列，要使用它必须包含下述代码：
```
#include<string>
using std::string;
```
初始化 string 对象的方式：
```
string s1; //默认初始化，s1 是一个空串
string s2(s1); //s2 是 s1 的副本
string s2=s1; //同上
string s3("value"); //直接初始化
string s3="value"; //拷贝初始化
string s4(n,'c'); //把 s4 初始化为由连续 n 个字符 c 组成的串
```
string 的操作：
```
os<<s; //把 s 写到输出流 os 中，返回 os
is>>s; //从 is 中读取字符串赋给 s，忽略空白字符，返回 is
getline(is,s) //从 is 中读取一行赋值给 s，保留空格，返回 is
s.empty() //s 为空返回 true,否则返回 false
s.size() //返回 s 中字符的个数
s[n] //返回 s 中第 n 个字符的引用，位置 n 从 0 记起
s1+s2 //返回 s1 和 s2 连接后的结果
s1=s2 //用 s2 的副本代替 s1 中原来的字符
s1==s2 //相等性判断区分大小写，要求所含字符完全一样
s1!=s2
<,<=,>,>= //按字典中的顺序进行比较，且区分大小写
```
该对象的比较规则如下：
1. 如果两个 string 对象的长度不同，而且较短 string 对象的每个字符都与较长 string 对象对应位置上的字符相同，就说较短 string 对象小于较长 string 对象
2. 如果两个 string 对象在某些对应的位置上不一致，则 string 对象比较的结果其实是 string 对象中第一对相异字符比较的结果

```
string word;
while(std::cin>>word)                //一直读到文件末尾
     std::cout<<word<<std::endl; //逐个输出单词，每个单词后面紧跟一个换行
string line;
while(getline(std::cin,line))        //每次读入一整行
     if(!line.empty())               //遇到空行直接跳过
        std::cout<<line<<std::endl;
```
编程时常常需要把表达式的值赋给变量，这就要求在声明变量时清楚知道表达式的类型，然而这并不是那么容易，C++11 新标准提供了 auto 类型说明符，用它就能让编译器替我们去分析表达式所属的类型，显然，auto 定义的变量必须有初始值
一条声明语句中只能有一个数据类型：
```
auto i=0,*p=&i; //正确：i 是整数，p 是整型指针
auto sz=0,pi=3.14; //错误：sz 和 pi 的类型不一致
```
引用是既有对象的另一个名字（别名）。**绑定**是令某个名字与给定的实体关联在一起，使用该名字也就是使用该实体，例如引用就是将某个名字与某个对象绑定在一起
如何让 devcpp 支持 C++11：工具——编译选项——在【编译时加入以下命令】处勾选——在下面的白框中输入-std=c++11

4.每输入一行，将其转为大写字母并按每行一个单词的形式输出.Cpp

```
#include<string>
#include<iostream>
#include<cctype>
using namespace std;
int main(){
    string word;
    while(cin>>word)
    {
       for(auto &c:word)      //注意：c 是引用，它被依次绑定到序列的每个元素上
          c=toupper(c);                //改变 c 绑定的字符的值
       cout<<word<<endl;
    }
    return 0;}
```

如果你想处理 string 对象中的每一个字符，你可以使用 C++11 提供的范围 for 语句，这种语句遍历给定序列中的每个元素并对序列中的每个值执行某种操作，其语法形式是：

for(*declaration:expression*)

其中 expression 部分是一个对象，用于表示一个序列，declaration 部分负责定义一个变量，该变量将被用于访问序列中的基础元素，每次迭代，变量会被初始化为序列中的下一个元素值。如果想要改变 string 对象中的字符值，必须把循环变量定义成引用类型

```
    string line;
    while(getline(cin,line))
        for(auto c:line)
           if(!ispunct(c))              //如果 c 不是标点符号
              cout<<c;                 //输出 c
    cout<<endl;                     //每输入一行，删除标点符号
```

当一个函数用它自己来定义时就称为是递归的，递归与循环的区别在于：虽然我们定义一个函数用的是这个函数本身，但是我们并没有用函数本身定义该函数的一个特定实例。换句话说，通过使用 f(5)来得到 f(5)的值才是循环的，通过使用 f(4)得到 f(5)的值不是循环的，除非 f(4)的求值又要用到对 f(5)的计算

递归的 4 条基本法则是：

1. 基准情形：必须总有某些基准情形，它无须递归就能解出
2. 不断推进：对于那些需要递归求解的情形，每一次递归调用都必须要使求解状况朝接近基准情形的方向推进

考虑一本大词典，词典中的词都是用其他的词定义的。当我们查一个单词的时候，我们不理解对该词的解释，于是我们不得不再查出现在解释中的一些词。而对这些词解释中的某些词我们又不理解，因此我们还要继续这种搜索。因为词典是有限的，所以实际上，要么我们最终查到一处，明白此处解释中所有的单词(从而理解这里的解释，并按照查找的路径回头理解其余的解释)；要么我们发现这些解释形成一个循环，无法明白其中的意思。

理解这些单词的递归策略如下：如果我们知道一个单词的含义，那么就算我们成功；否则，我们就在词典里查找这个单词。如果我们理解对该词解释中的所有的单词，那么又算我们成功；否则，递归地查找一些我们不认识的单词来“算出”对该单词解释的含义。如果词典编纂地完美无瑕，那么这个过程就能够终止；如果其中一个单词没有查到或是形成循环解释，那么这个过程则循环不定。

3. 设计法则：假设所有的递归调用都能运行
4. 合成效益法则：在求解一个问题的同一实例时，切勿在不同的递归调用中做重复性的工作

使用递归来计算诸如斐波那契数之类简单数学函数值的想法一般来说不是一个好主意，其道

理正是根据第 4 条法则

```
long int fib(int n){
    if(n<=1)
       return 1;
    else
       return fib(n-1)+fib(n-2); /*第 3 行*/}
```

这个程序之所以缓慢，是因为存在大量多余的工作要做。在第 3 行上的第 1 次调用即 fib(n-1)实际上计算了 fib(n-2)，这个信息被抛弃而在第 3 行上的第 2 次调用时又重新计算了一遍，抛弃的信息量递归地合成起来导致了按指数级增长的运行时间。这或许是格言“计算任何事情不要超过一次”的最好例子，但这不应该使你被吓得远离递归而不敢用它

定义：如果存在正常数 c 和n_0使得当$\mathrm{N} \geq n_0$时$\mathrm{T(N)} \leq \mathrm{cF(N)}$，则记为$\mathrm{T(N)} = \mathrm{O(F(N))}$

上面定义了分析算法时间复杂度的大 O 记号，一个语句的频度是指该语句在算法中被重复执行的次数，算法中所有语句的频度之和记为 T(N)，它是算法问题规模 N 的函数，时间复杂度主要分析 T(N)的数量级，算法中的基本运算(最深层循环内的语句)的频度 F(N)与 T(N)同数量级。因此算法的时间复杂度记为$\mathrm{T(N)} = \mathrm{O(F(N))}$ ，当我们说$\mathrm{T(N)} = \mathrm{O(F(N))}$时，我们是在保证函数 T(N)是在以不快于 F(N)的速度增长

需要掌握的重要结论：如果$\mathrm{T_1(N)} = \mathrm{O}\big(\mathrm{F(N)}\big)$且$T_2(N) = O(G(N))$，那么

$T_1(N) + T_2(N) = \max(O\big(F(N), G(N)\big))$　　$T_1(N) * T_2(N) = O(F(N) * G(N))$

对任意常数 k，$log^k N = O(N)$，它告诉我们对数增长得非常缓慢

一次 for 循环的运行时间最多是循环内语句(包括测试)的运行时间乘以迭代次数

在一组嵌套循环内部的一条语句总的运行时间为该语句的运行时间乘以该组所有的 for 循环的大小的乘积

43.最大子序列和问题.C

```
#include<stdio.h>
int MaxSubsequenceSum(const int A[],int n);
int main(){
    const int A[6]={-2,11,-4,13,-5,-2};
    const int B[8]={4,-3,5,-2,-1,2,6,-2};
    int res1,res2;
    res1=MaxSubsequenceSum(A,6);
    res2=MaxSubsequenceSum(B,8);
    printf("%d\n%d",res1,res2);      /*输出结果是 20 和 11*/
    return 0;
}
```

分治法(devide and conquer)：把问题分成两个大致相等的子问题，然后递归地对它们求解，这是“分”部分，“治”阶段把两个子问题的解合并到一起并可能再做些少量的附加工作，最后得到整个问题的解。在我们的例子中，最大子序列和可能在 3 处出现：整个出现在输入数据的左半部、或者右半部、或者跨越中部从而占据左右两半部分。前两种情况可以递归求解，第三种情况可以通过求出前半部分的最大和以及后半部分的最大和，然后把它们加起来而得到。

```
static int MaxSubSum(const int A[],int left,int right){
    int MaxLeftSum,MaxRightSum;
    int MaxLeftBorderSum,MaxRightBorderSum;
    int LeftBorderSum,RightBorderSum;
    int center,i;
    int max3(int a,int b,int c);
    if(left==right)              //处理基准情况
       if(A[left]>0)
          return A[left];
```

```
        else
           return 0;
    center=(left+right)/2;
    MaxLeftSum=MaxSubSum(A,left,center);           //执行两次递归调用
    MaxRightSum=MaxSubSum(A,center+1,right);
    MaxLeftBorderSum=0;LeftBorderSum=0;
    for(i=center;i>=left;i--)
    {                                  //计算分界线两边的两个最大和
        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);}
int MaxSubsequenceSum(const int A[],int n){return MaxSubSum(A,0,n-1);}
int max3(int a,int b,int c){
    int max=a;
    if(b>max) max=b;
    if(c>max) max=c;
    return max;}
```

时间复杂度为$O(N\log N)$。

寻找二叉树中两个结点的最低公共祖先：

```
struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode(int x) : val(x), left(NULL), right(NULL) {}
};
TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q)
{
    if(!root||root==p||root==q)
        return root;
    TreeNode *left=lowestCommonAncestor(root->left,p,q);
    TreeNode *right=lowestCommonAncestor(root->right,p,q);
    return !left?right:!right?left:root;
}
```

下面编写一个更高效率的幂运算函数：

```
int iseven(int n){return n%2==0;};
long int pow(long int x,unsigned int n){
    if(n==0)                            //基准情形
       return 1;
    if(n==1)
       return x;
    if(iseven(n))                       //偶数判定
       return pow(x*x,n/2);
    else
       return pow(x*x,n/2)*x;};
```

使用该算法计算 x 的 62 次方只需要用到 9 次乘法

对于除空表之外的任何表，我们说A_{i+1}后继A_i并称A_{i-1}前驱A_i ，对表的所有操作都可以通过使用数组来实现，虽然数组是动态指定的，但还是需要对表的大小的最大值进行估计，通常需要估计得大一些，从而浪费大量的空间。因为插入和删除需要线性时间以及表的大小还需要事先已知，所以简单数组一般不用来实现表这种结构

为了避免这个问题，我们需要允许表可以不连续存储，否则表的部分或全部需要整体移动

链表由一系列不必在内存中相连的结构组成，每一个结构均含有表元素和指向包含该元素后继元的结构的指针，我们称之为 next 指针，最后一个单元的 next 指针指向 NULL(表示什么也不指向，即空指针)

双向链表：只需在结点单元上附加一个域，使之包含指向前驱单元的指针即可

循环链表：让最后的结点单元反过来指向第一个结点单元

```
#include<stdlib.h>
#include<stdio.h>
#ifndef _List_h
typedef int ElementType;
struct Node;                                                    //结点
typedef struct Node *PtrToNode;          //PtrToNode：指向结点的指针
typedef PtrToNode List;                  //表
typedef PtrToNode Position;              //表示表中元素位置的指针
struct Node{
    ElementType Element;                 //数据域
    Position next;                       //指针域
};
/*测试一个链表是否是空表的函数*/
int IsEmpty(List L){                 //L 是指向表头(哑结点,header)的指针
    return L->next==NULL;};
/*测试当前位置是否是链表的末尾的函数*/
int IsLast(Position p,List L){
    return p->next==NULL;};
/*Find 例程：返回某个元素在表中的位置*/
Position Find(ElementType x,List L){
    Position p;
    p=L->next;
    while(p!=NULL&&p->Element!=x)                //检查数据域是否对应
        p=p->next;                          //如果不对应，继续指向后继结点检查
    return p;};
/*FindPrevious 例程：返回指向含有数据 x 的结点的前驱结点的指针*/
Position FindPrevious(ElementType x,List L){
    Position p;
    p=L;                                        //初始化，从表头开始
    while(p->next!=NULL&&p->next->Element!=x)
        p=p->next;
    return p; };
/*链表的删除例程：删除表 L 中的某个结点 x*/
void Delete(ElementType x,List L)
{
    Position p,tmpcell;           //指针 1 前驱 指针 2 结点 x 指针 3 后继
    p=FindPrevious(x,L);                      //指针 1 赋值给 p
    if(!IsLast(p,L))
    {
        tmpcell=p->next;                    //指针 2 赋值给 tmpcell
        p->next=tmpcell->next;              //把指针 3 的出发点移动到前驱上
        free(tmpcell);
```

```
            //tmpcell 正在指向的地址没变，但在该地址的数据此时已经没有定义了
    }       //理解指针赋值：把等号前的指针指向等号后的指针指向的地方
};          //后面的指针将会指向不确定的区域
/*链表的插入例程：*/
void Insert(ElementType x,List L,Position p)
{
    Position tmpcell;
    tmpcell=malloc(sizeof(struct Node));
                                //新建一个结点，并把 tmpcell 指向这个结点
    if(tmpcell==NULL)               //malloc 函数只给指针分配一个空间
       printf("error:out of space\n");
    tmpcell->Element=x;                         //给数据域赋值
    tmpcell->next=p->next;
    p->next=tmpcell;
};
/*删除整个链表的例程*/
void DeleteList(List L){
    Position p,tmp;
    p=L->next; //Header assumed
    L->next=NULL;
    while(p!=NULL)
    {
        tmp=p->next;
        free(p);
        p=tmp;
    }
};
#endif
int main(){return 0;}
```

下面来看一个使用链表的例子：多项式 ADT，令$F(X)=\sum_{i=0}^{N} A_i X^i$ ，如果大部分系数非零，可以用一个简单数组来存储这些系数

多项式 ADT 的数组实现：

```
#include<stdio.h>
#define maxdegree 20
#define Max(a,b) ((a)>(b)?(a):(b))
/*多项式的类型声明*/
typedef struct{
    int Coeff[maxdegree+1];
    int HighPower;
} *Poly;
/*将多项式初始化为 0 的例程*/
void ZeroPolynomial(Poly p)
{
    int i;
    for(i=0;i<=maxdegree;i++)
        p->Coeff[i]=0;
    p->HighPower=0;
};
/*两个多项式相加的例程*/
void AddPolynomial(const Poly p1,const Poly p2,Poly psum)
{
    int i;
    ZeroPolynomial(psum);
    psum->HighPower=Max(p1->HighPower,p2->HighPower);
```

```
    for(i=psum->HighPower;i>=0;i--)
        psum->Coeff[i]=p1->Coeff[i]+p2->Coeff[i];
};
/*两个多项式相乘的例程*/
void MultPolynomial(const Poly p1,const Poly p2,Poly pProd)
{
    int i,j;
    ZeroPolynomial(pProd);
    pProd->HighPower=p1->HighPower+p2->HighPower;
    if(pProd->HighPower>maxdegree)
        printf("error:exceeded array size.\n");
    else
        for(i=0;i<=p1->HighPower;i++)
           for(j=0;j<=p2->HighPower;j++)
              pProd->Coeff[i+j]+=p1->Coeff[i]*p2->Coeff[j];
};
int main(){return 0;}
```

栈是限制插入和删除操作只能在一个位置上进行的表，该位置叫做栈顶。可以把栈想象为手枪的弹夹，栈的元素是子弹。

栈有两种实现方法：指针实现和数组实现，如果用链表(指针)实现栈，栈顶就是表的头部；如果用数组实现栈，栈顶是数组的尾部

下面来看栈的链表实现：

```
#include<stdio.h>
#include<stdlib.h>
#ifndef _stack_h
typedef int ElementType;
struct Node;
typedef struct Node *PtrToNode;
typedef PtrToNode stack;
struct Node{
    ElementType element;
    PtrToNode next;};
/*测试栈是否是空栈的例程*/
int IsEmpty(stack S){return S->next==NULL;};
/*push 进栈的例程*/
void push(ElementType x,stack S)
{                  //S 指向表头
    PtrToNode tmpcell;
    tmpcell=malloc(sizeof(struct Node));
    if(tmpcell==NULL)
       printf("error:out of space!!!");
    else
    {
        tmpcell->element=x;
        tmpcell->next=S->next;
        S->next=tmpcell;
    }
};
/*返回栈顶元素的例程*/
int top(stack S){
    if(!IsEmpty(S))                        /*栈顶后继表头，我们保留了表头*/
       return S->next->element;
    printf("error:empty stack\n");
    return 0;                              /*使用返回值来防止编译警告*/
```

```
};
/*从栈弹出元素的例程*/
void pop(stack S){
    PtrToNode firstcell;
    if(IsEmpty(S))
       printf("error:empty stack\n");
    else
    {
        firstcell=S->next;
        S->next=S->next->next;
        free(firstcell);
    }
};
/*创建一个空栈的例程*/
void MakeEmpty(stack S){
    if(S==NULL)                              /*要先有一个表头*/
       printf("error:must use CreatStack first\n");
    else
       while(!IsEmpty(S))
          pop(S);                             /*弹出所有结点直到栈空*/
};
stack CreatStack(void){
    stack S;
    S=malloc(sizeof(struct Node));
    if(S==NULL)
       printf("error:out of space!!!");
    S->next==NULL;                       /*创建一个表头即可*/
    MakeEmpty(S);
    return S;
};
#endif
int main(){return 0;}
```

栈的链表实现的缺点主要在于对 malloc 和 free 的调用的开销是昂贵的。有的缺点可以通过使用第二个栈来避免，该第二个栈初始时为空栈，当一个单元从第一个栈弹出时，它只是被放到了第二个栈中，此后，当第一个栈需要新的单元时，它首先去检查第二个栈

栈的数组实现是比链表实现更流行的方式：

```
#include<stdio.h>
#include<stdlib.h>
#ifndef _stack_h
#define empty_to_s -1
#define min_stack_size 5
#define fatalerror() printf("error:out of space!!!\n");
#define emptyerror() printf("error:empty stack\n");
typedef int ElementType;
struct StackRecord{        //声明一个记录栈的信息的结构体，它包含的信息有
    int capacity;                            //栈的容量，即数组大小
    int TopOfStack;                          //栈顶的数组下标
    ElementType *array;};                    //指向这个数组的指针
typedef struct StackRecord *stack;           //S 是指向这个结构体的指针
/*创建一个空栈的例程*/
void MakeEmpty(stack S){S->TopOfStack=empty_to_s;};
/*检测一个栈是否为空的例程*/
int IsEmpty(stack S){return S->TopOfStack==empty_to_s;};
/*创建栈的例程*/
```

```
stack CreatStack(int max_elements)
{                                                    //数组的最大容量
    stack S;
    if(max_elements<min_stack_size)
       printf("error:stack size is too small\n");
    S=malloc(sizeof(struct StackRecord));
    if(S==NULL)
       fatalerror();
    S->array=malloc(sizeof(ElementType)*max_elements); //创建数组
    if(S->array==NULL)
       fatalerror();
    S->capacity=max_elements;                      //设定数组的容量
    MakeEmpty(S);   //把栈设为空，相当于造了一个空弹夹，里面还没有放子弹
    return S;
};
/*释放栈的例程*/
void DisposeStack(stack S){
    if(S!=NULL)
    {
        free(S->array);                       //先释放栈数组
        free(S);                              //后释放栈结构体
    }
};
/*检测栈是否已经放满的例程*/
int IsFull(stack S){return S->TopOfStack==S->capacity;};
/*进栈的例程*/
void push(ElementType x,stack S){
    if(IsFull(S))
       printf("error:full stack\n");
    else
       S->array[++S->TopOfStack]=x;};
/*返回栈顶元素的例程*/
ElementType top(stack S){
    if(!IsEmpty(S))
       return S->array[S->TopOfStack];
    emptyerror();
    return 0;};                           //返回 0 用于防止编译器警告
/*从栈弹出元素的例程*/
void pop(stack S){
    if(IsEmpty(S)){
       emptyerror();}
    else
       S->TopOfStack--;
};
/*给出栈顶元素并从栈弹出的例程*/
ElementType topANDpop(stack S){
    if(!IsEmpty(S))
        return S->array[S->TopOfStack--]; //这条语句完美地体现了 C 语言的简洁
    emptyerror();
    return 0;
};
#endif
int main(){return 0;}
```

来看一个栈的应用；平衡符号，为了检测程序的语法错误，比如未配对的花、方、圆括号(开

放符号)，我们设计一个简单的算法，它需要用到一个栈：
做一个空栈。读入字符直到文件尾。如果(读到的)字符是一个开放符号，则将其压入栈中。如果(读到的)字符是一个封闭符号，则：当栈空时报错，当栈非空时将栈元素弹出。如果弹出的符号不是对应的开放符号，则报错。在文件尾，如果栈非空则报错。
再来看一个应用：中缀表达式转逆波兰表达式
例如 a+b*c+(d*e+f)*g 转为 a b c * + d e * f + g * +
读取中缀表达式，当读到一个数时，立即把它丢到输出中。读到运算符和左圆括号时，把它们压入栈中。如果读到一个右圆括号，就把栈中元素弹出，并将弹出的符号一个接一个输出直到我们遇到一个和右括号对应的左括号，但是这个左括号只被弹出而不输出；如果读到任何一个其他符号，那么我们从栈中依次弹出优先级不低于这个符号的运算符，然后把这个符号压入栈中。有一个例外：除非是在处理右括号的时候，否则绝不弹出左括号，对于这种操作，优先级从低到高为+ * (。最后，如果我们读到中缀表达式的末尾，就把栈中所有符号依次弹出并输出从而清空栈。
递归的精髓在于能否将原始问题转换成属性相同但规模较小的问题，在递归调用的过程中，计算机系统为每一层调用的返回点、局部变量、传入实参等开辟了递归工作栈来进行数据存储，所存储的信息被称为活动记录或栈帧，递归次数过多容易造成**栈溢出**(stackoverflow)在不进行栈溢出检测的语言和系统中，程序将会崩溃而没有明显的说明！
队列也是表，然而使用队列时插入在一端进行而删除在另一端进行。入队：在队尾插入一个元素；出队：删除或返回队头元素

```
#include<stdio.h>
struct QueueRecord;                          //记录队列信息的结构体
typedef struct QueueRecord *Queue;              //指向结构体的指针
typedef int ElementType;
struct QueueRecord{
    int Capacity;                            //队列数组的大小
    int Front;                             //指向队头元素的指针
    int Rear;                            //指向队尾元素的指针
    int Size;                        //实际存在于队列中的元素个数
    ElementType *Array;};                    //指向队列数组的指针
/*测试队列是否为空的例程*/
int IsEmpty(Queue Q){return Q->Size==0;};
/*测试队列是否已满的例程*/
int IsFull(Queue Q){return Q->Size==Q->Capacity;}
/*构造空队列的例程*/
void MakeEmpty(Queue Q)
{
    Q->Size=0;
    Q->Front=1;
    Q->Rear=0;
};
/*入队的例程*/
static int Succ(int value,Queue Q)
{
    if(++value==Q->Capacity)
       value=0;
    return value;
};
void Enqueue(ElementType x,Queue Q)
{
    if(IsFull(Q))
```

```
        printf("error:full queue\n");
    else
    {
        Q->Size++;
        Q->Rear=Succ(Q->Rear,Q);
        Q->Array[Q->Rear]=x;
    }
};
int main(){return 0;}
```

若要编写出队例程，置返回值为 Queue[Front]，Size 减 1，然后使 Front 增 1

为了不浪费队列数组的空闲位置，简单的解决办法是，只要 Front 或 Rear 到达数组的尾端，它就又绕回到开头。这叫做循环数组实现。实现回绕所需的附加代码是极小的(虽然它可能使运行时间加倍)，如果 Front 或 Rear 增 1 使得超越了数组，那么其值就要重置为数组的第一个位置。有两个问题要注意：检测队列是否为空是很重要的；某些程序员使用不同的方法来表示队头和队尾，如果你要修改就必须仔细

尾递归与普通递归的区别在于，尾递归中整个递归调用占据了整个 return 后面的部分，而不是表达式的一部分。由于它在整个方法的最后才被调用，之前函数压栈保留的所有局部变量等等信息都不影响下一次递归调用。本次调用之后栈被清空，栈可以重复利用，所以尾递归的效率更高。

给定整数 n，存储值 1 ... n 的结构唯一的 BST（二叉搜索树）有多少个？

只要将每个数字作为根结点，然后左边部分×右边部分就是答案

```
    public int numTrees(int n) {
        int[] G=new int[n+1];
        G[0]=G[1]=1;
        for(int i=2;i<=n;i++){
            for(int j=1;j<=i;j++){
                G[i]+=G[j-1]*G[i-j];
            }
        }
        return G[n];
    }
```

表、栈、队列在全部计算机科学中是最基本的 3 个数据结构

定义树的一种自然的方式是递归的方法，一棵树是一些节点的集合。这个集合可以是空集；若非空，则一棵树由称作根的结点 r 和 0 个或多个非空的子树组成，这些子树中每一棵的根都被来自 r 的一条有向的边所连接。每一棵子树的根叫做根 r 的儿子，而 r 是每一棵子树的根的父亲。一棵树是 N 个节点和 N-1 条边的集合，其中的一个节点是根。每条边都将某个节点连接到它的父亲，而除去根节点外每一个节点都有一个父亲。没有儿子的节点称为树叶。从节点n_1到n_k的路径定义为节点$n_1, n_2, \ldots, n_k$的一个序列，使得对于$1 \le \mathrm{i} < \mathrm{k}$，节点$n_i$是$n_{i+1}$的父亲。这个路径的长为该路径上边的条数，即 k-1。对任意节点，它的深度定义为从根到该节点的唯一路径的长，它的高定义为从该节点到一片树叶的最长路径的长

二叉树是一棵树，其中每个节点都不能有多于 2 个的儿子，它的子树有左右之分，且次序不能颠倒，就算只有一个孩子，孩子的左右次序也是确定的；二叉树可以为空。用树表示算术表达式的方法是把操作数放在树叶上，而把运算符放在其他节点上。表达式树不一定是二叉树。二叉查找树也叫二叉排序树，使一棵二叉树成为二叉查找树的性质是：对于树中的每个节点 X，它的左子树中**所有**关键字值小于 X 的关键字值，而它的右子树中所有关键字值大于 X 的关键字值。这意味着，该树所有的元素可以用某种统一的方式排序

```
#include<stdio.h>
#include<stdlib.h>
struct TreeNode;
```

```
typedef struct TreeNode *Position;                      //指示元素位置
typedef struct TreeNode *SearchTree;                     //指向根节点
typedef int ElementType;
struct TreeNode{
    ElementType element;                                //数据域
    SearchTree left;                                    //指向左子树
    SearchTree right;};                                  //指向右子树
/*建立一棵空树的例程*/
SearchTree MakeEmpty(SearchTree T)
{
    if(T!=NULL)
    {
        MakeEmpty(T->left);
        MakeEmpty(T->right);
        free(T);
    }
    return NULL;
};
/*Find 例程：返回指向树 T 中具有关键字 x 的节点的指针*/
Position Find(ElementType x,SearchTree T)
{
    if(T==NULL)                                     //首先测试是否为空树
       return NULL;
    else if(x<T->element)                       //对左右子树的递归调用
       return Find(x,T->left);
    else if(x>T->element)
       return Find(x,T->right);
    else
       return T;
};
/*FindMin 例程：返回树中最小元的位置，递归实现*/
Position FindMin(SearchTree T)
{
    if(T==NULL)                                         //从根开始
       return NULL;
    else if(T->left==NULL)                      //终止点是最小的元素
       return T;
    else                                        //只要有左儿子就向左进行
       return FindMin(T->left);
};
/*FindMax 例程：返回树中最大元的位置，非递归实现*/
Position FindMax(SearchTree T){
    if(T!=NULL)
       while(T->right!=NULL)
         T=T->right;
    return T;
};
/*插入元素到二叉查找树的例程*/
SearchTree Insert(ElementType x,SearchTree T){
    if(T==NULL)
    {
        T=malloc(sizeof(struct TreeNode));
        if(T==NULL)
           printf("error:out of space!!!\n");
        else
```

```
        {
            T->element=x;
            T->left=T->right=NULL;
         }
    }
    else if(x<T->element)
         T->left=Insert(x,T->left);
    else if(x>T->element)
         T->right=Insert(x,T->right);
    return T;
};
/*二叉查找树的删除例程*/
SearchTree Delete(ElementType x,SearchTree T){
    Position tmpcell;                                    //分 3 种情况
    if(T==NULL)                              //待删除的节点没有儿子(树叶)
       printf("error:element not found.\n");
    else if(x<T->element)
       T->left=Delete(x,T->left);                          //向左进行
    else if(x>T->element)
       T->right=Delete(x,T->right);                       //向右进行
                              //执行以上递归可以找到待删除的树叶节点并删除
    else if(T->left&&T->right)
    {                 //待删除的节点有 2 个儿子
        tmpcell=FindMin(T->right);               //用其右子树的最小数据
        T->element=tmpcell->element;                //代替该节点的数据
        T->right=Delete(T->element,T->right);  //并递归地删除那个节点
    }
    else
    {                                        //待删除的节点有 1 个儿子
        tmpcell=T;    //待删除的节点可以在其父节点调整指针绕过该节点后被删除
        if(T->left==NULL)
           T=T->right;
        else if(T->right==NULL)
            T=T->left;
        free(tmpcell);
    }
    return T;
};
int main(){return 0;}
```

下面来看 Python 的使用方法:下载 Python——windows 下选择开始菜单——所有程序——Python2.7——IDLE(Python 自带的 IDE)——File——New File——SaveAs 命名并保存编辑窗口中先 Ctrl+S 保存然后点击 Run 即可运行程序，也可以直接在 IDLE 中写代码并按回车键运行，代码和运行结果会同时展现

1.Hello world 程序.py

```
>>> print 'hello world!'
hello world!
```

2.画出 Sierpinski 三角形.py

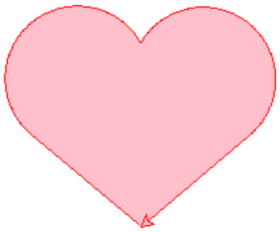
```
from turtle import*                    #使用 Python 自带的海龟绘图库 turtle
def huhu(l,d):                                  #定义一个函数
    pencolor("blue")                            #设置画笔颜色
    speed(10)                       #设置移动速度为 10，范围是 0~10 之间的整数
    if d>1:dot()                      #mark position to better see recursion
```

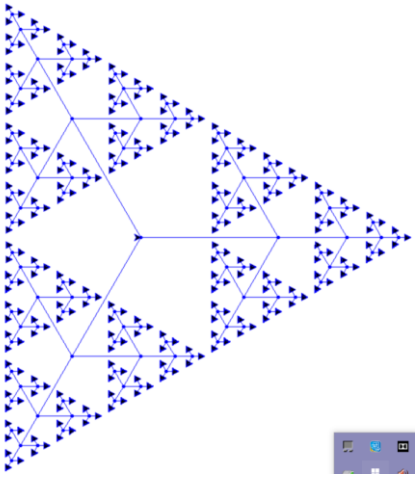
```
    if d==0:                                    #base case
        stamp()                                 #在当前位置复制一个海龟形状
    else:
        forward(l)                              #海龟移动命令：沿当前方向移动 l
        huhu(l/2,d-1)                           #递归调用
        backward(l)                             #沿与当前方向相反方向移动 l
        left(120)                               #左转 120 度
        forward(l)
        huhu(l/2,d-1)
        backward(l)
        left(120)
        forward(l)
        huhu(l/2,d-1)
        backward(l)
        left(120)
huhu(200,6)
done()                                  #运行这个程序可以让你清楚地观察到递归调用的机制
```

3.画心形.py

```
from turtle import*
def curvemove():                                #定义一个画图用的函数
    for i in range(200): #for 循环：i 从 1 到 199
        right(1)
        forward(1)
color('red','pink')                             #设置画笔颜色和填充颜色
begin_fill()
left(140)
forward(111.65)
curvemove()
left(120)
curvemove()
forward(111.65)
end_fill()                                      #在要填充图形后调用
done()
```

本书使用的Python版本为2.7





脚本语言是不需要编译的，即这个 script(剧本)是不用拍成电影的，一步一步“解释”着执行就可以(也叫“解释型语言”)。它和编译型语言都属于计算机的高级语言

Python 是一种面向对象的、免费开源的、广义的脚本语言。目前已经越来越流行，它的简单高效让程序员从C/C++的苦海中脱离出来，并喊出了“人生苦短，我用Python”的口号

Python 目前有两个主要版本：Python2 和 Python3，两者有细微差别

现在难免觉得 C 有些无聊，但是还要继续，掌握了数据结构与算法就掌握了编程的精髓，之后再切换语言就是相当自由的事情了，这里只是对 Python 做一个介绍，而你会发现，既然学过 C 语言，第一次遇到上面的 Python 程序你基本上也能看懂大意

有限状态机(finite state machine)是一个数学概念，如果把它运用于程序中，可以发挥很大的作用。它是一种协议，用于有限数量的子程序(“状态”)的发展变化。每个子程序进行一些处理并选择下一种状态(通常取决于下一段输入)

FSM 可以用作程序的控制结构，它对于那些基于输入的在几个不同的可选动作中进行循环的程序尤其适合。投币售货机就是这样一个例子，它具有接受硬币、选择商品、发送商品、找零钱等数种状态。它的输入是硬币，输出是待售商品

它的基本思路是用一张表保存所有可能的状态，并列出进入每个状态时可能执行的所有动作，其中最后一个动作就是计算下一个应该进入的状态。你从一个初始状态开始，在这一过程中翻译表可能会告诉你进入了一个错误的状态，表示一个预期之外的或错误的输入。你不停地在各种状态间进行转换，直到达到结束状态

C 语言中可以用函数指针数组来表达 FSM，可以这样声明：`void (*state[MAX_STATES])()`

如果知道了函数名，就可以像这样对数组初始化

`extern int a(),b(),c(),d();  int (*state[])()={a,b,c,d};`

可以通过数组中的指针来调用函数： `(*state[i])();`

所有的函数必须接受同样的参数，并返回同种类型的返回值(除非你把数组元素做成一个联合)，甚至可以去掉指针形式，把调用写成 `state[i]();`

C 没有规定花括号必须换行，程序员中也有换行党和不换行党，这两方我谁也不支持，如果你想让代码可读性提高，或者工资按照代码行数来算，那就换行好了，别那么固执

注意：Java 是规定了花括号不能换行的！

下面来写一个换行了的程序，打印出来确实有点浪费纸，但是能更容易地看懂了

44.用有限状态机分析 C 语言的声明(cdecl 程序的完整版本).C

这里有一个设计方案：主要的数据结构是一个栈，我们从左向右读取，把各个标记依次压入栈，直到读到标识符为止。然后我们继续向右读入一个标记，接着，观察标识符左边的那个标记(需要从栈中弹出)

```
#include<stdio.h>
#include<string.h>
#include<ctype.h>
#define MAXTOKENS 100
#define MAXTOKENLEN 64
enum type_tag{IDENDIFIER,QUALIFIER,TYPE};       //标识符、限定符、类型
struct token
{
    char type;
    char string[MAXTOKENLEN];
};
int top=-1;
struct token stack[MAXTOKENS];              //保存第一个标识符之前的所有标记
struct token this;                                  //保存刚读入的那个标记
#define pop stack[top--]
#define push(s) stack[++top]=s
enum type_tag classify_string(void)
{                                                           //推断标识符的类型
    char *s=this.string;
    if(!strcmp(s,"const"))                              //查看当前的标记
    {
        strcpy(s,"read-only");
        return QUALIFIER;
    }
    if(!strcmp(s,"volatile"))  return QUALIFIER;
    if(!strcmp(s,"void"))      return TYPE;
    if(!strcmp(s,"char"))      return TYPE;
    if(!strcmp(s,"signed"))    return TYPE;
    if(!strcmp(s,"unsigned"))  return TYPE;
    if(!strcmp(s,"short"))     return TYPE;
```

```
    if(!strcmp(s,"int"))       return TYPE;
    if(!strcmp(s,"long"))      return TYPE;
    if(!strcmp(s,"float"))     return TYPE;
    if(!strcmp(s,"double"))    return TYPE;
    if(!strcmp(s,"struct"))    return TYPE;
    if(!strcmp(s,"union"))     return TYPE;
    if(!strcmp(s,"enum"))      return TYPE;
    return IDENDIFIER;
}
void gettoken(void)
{                                                   //读入下一个标记，保存在 this 中
    char *p=this.string;
    while((*p=getchar())==' ');                         //略过所有空格
    if(isalnum(*p))                                 //如果是字母数字组合
    {
        while(isalnum(*++p=getchar()));     //在标识符中读入 A~Z,1~9 字符
        ungetc(*p,stdin);
        *p='\0';
        this.type=classify_string();
        return;
    }                                               //否则它一定是一个单字符标记
    this.string[1]='\0';                           //用一个 nul 结束 this.string
    this.type=*p;                                   //this.type=该标记
    return;
}                //下面构造一个有限状态机，每个函数的参数和返回值类型都是 void
void initialize(),get_array(),get_params(),get_lparen(),get_ptr_part(),get_type();
void (*nextstate)(void)=initialize;                             //初始状态
int main()
{
    while(nextstate!=NULL)
        (*nextstate)();              //在不同的状态间转换，直到指针指 NULL
    return 0;
}
void initialize()                                           //读到第一个标识符
{
    gettoken();                                            //调用 gettoken
    while(this.type!=IDENDIFIER)                         //直到遇见一个标识符
    {
        push(this);                                       //并把标记压入到栈中
        gettoken();
    }
    printf("%s is ",this.string);                        //打印(标识符是)
    gettoken();                                              //继续调用 gettoken
    nextstate=get_array;
}
void get_array()
{
    nextstate=get_params;
    while(this.type=='[')          //当你读取[size]后，打印之，并继续向右读取
    {
        printf("array ");
        gettoken();                                             //一个数字或']'
        if(isdigit(this.string[0]))
        {                        //库函数 atoi：将字符串 s 转换为相应的整型数
            printf("0..%d ", atoi(this.string) - 1);
            gettoken();                                          //读取']'
        }
        gettoken();                                         //在']'之后读取
        printf("of ");
        nextstate=get_lparen;
    }
}
void get_params()
```

```
{
    nextstate=get_lparen;
    if(this.type=='(')
    {
        while(this.type!=')')
        {
            gettoken();                             //当读取越过右括号')'后
        }
        gettoken();
        printf("function returning ");
    }                                                       //打印"函数返回"
}
void get_lparen()
{
    nextstate=get_ptr_part;
    if(top>=0)                                  //如果栈里还有东西
    {
        if(stack[top].type=='(')                    //如果它是一个'('
        {
            pop;                                   //将其弹出
            gettoken();                  //并调用 gettoken，应该获得')'
            nextstate=get_array;
        }
    }
}
void get_ptr_part()
{
    nextstate=get_type;
    if(stack[top].type=='*')                    //当你从栈中读取'*'时
    {
        printf("pointer to ");                      //打印"指向...的指针"
        pop;                                       //并将其弹出栈
        nextstate=get_lparen;
    }
    else if(stack[top].type==QUALIFIER)
    {
        printf("%s ",pop.string);
        nextstate=get_lparen;
    }
}
void get_type()
{
    nextstate=NULL;
    while(top>=0)              //处理在读入标识符之前被放在栈中的所有标记
    {
        printf("%s ",pop.string);
    }                                                   //将其弹出栈并打印
    printf("\n");
}
```

举例：char *(*c[10])()

c is array 0..9 of pointer to function returning pointer to char

char (*(*x())[])()

x is function returning pointer to array of pointer to function returning char

cdecl 程序不能分析函数的参数，你可以把它们的声明单独拿出来分析

树中一个节点的子节点个数称为该节点的度，树中节点的最大度数称为树的度，度>0 的节点叫做分支节点，度为 0 的节点叫做树叶(终端节点)，节点的层次从根开始定义，根节点为第 1 层，树的高度是树中节点的最大层数。树的路径长度是从根到每一个结点的路径长度的总和。总结点数=总分支数+1，总分支数=所有结点的度数和，

向上取整符号：⌈a⌉表示比 a 大的最小整数，向下取整符号：⌊a⌋表示比 a 小的最大整数

AVL 树是带有平衡条件的二叉搜索树，一棵 AVL 树是其每个节点的左子树和右子树的高度最多相差 1 的二叉搜索树。定义节点左右子树的高度差为节点的平衡因子，则平衡二叉树的节点的平衡因子的值只能取-1,0,1

二叉树的 4 种遍历算法：先序遍历，中序遍历，后序遍历，层次遍历

先序遍历：根节点→左子树→右子树　　中序遍历：左子树→根节点→右子树

后序遍历：左子树→右子树→根节点　　这三种算法都是递归实现的

层次遍历：需要借助一个队列，先将根节点入队，然后出队，访问该节点，如果它有左子树，则将左子树的根节点入队；如果它有右子树，则将右子树的根节点入队。然后出队，访问出队的节点，如此循环，直到队列为空

B 树又称为多路平衡查找树，B 树中所有节点的孩子节点数的最大值为 B 树的阶，通常用 M 表示，一棵 M 阶 B 树或为空，或为满足以下特性的 M 叉树：

树中每个节点最多有 M 棵子树(即最多含有 M-1 个关键字)

如果根节点不是终端节点，则至少有 2 棵子树

除根节点外的所有非叶节点至少有⌈M/2⌉棵子树(即至少含有⌈M/2⌉-1 个关键字)

所有非叶节点的结构如下：

n	p_0	K_1	p_1	K_2	p_2	$\cdots..$	$\cdots..$	K_n	p_n

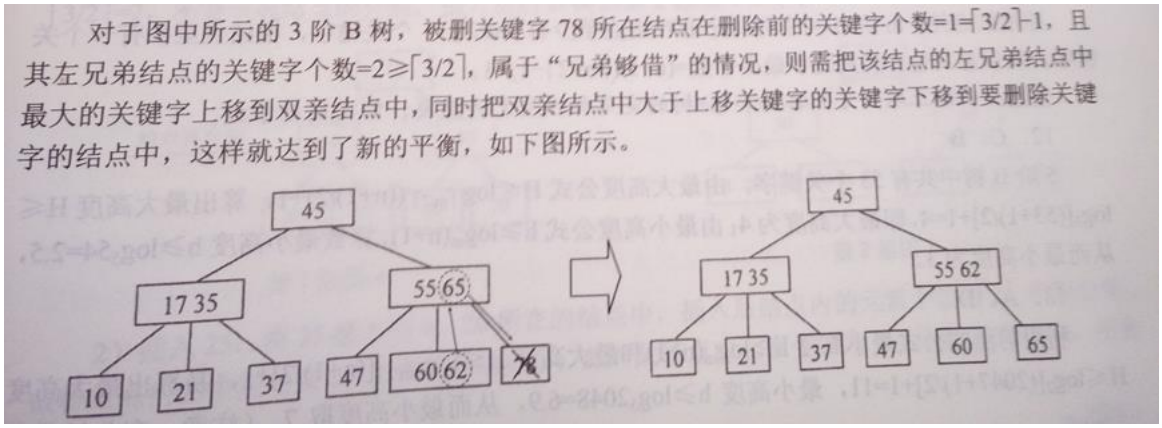
其中K_i为节点的关键字，且满足$K_1 < K_2 < \cdots < K_n$，p_i为指向子树根节点的指针，且指针p_{i-1}指向的子树中所有节点的关键字均小于K_i，p_i指向的子树中所有节点的关键字均大于K_i

n 为节点中关键字的个数(⌈M/2⌉-1<=n<=M-1)，所有的树叶都出现在同一层次上，并且不带信息。B 树是所有节点的平衡因子均为 0 的多路查找树

B 树的插入：首先定位，插入关键字后检查被插入节点内关键字的个数，如果插入之后节点内关键字个数大于 M-1，则必须对节点进行分裂，分裂的方法是：取一个新节点，将插入 key 后的原节点从中间位置将其中的关键字分为两部分，左部分留在原节点，右部分放到新节点中，中间位置的节点插入到原节点的父亲节点中，若此时导致其父亲节点中的关键字个数也超过了上限，则继续这种分裂操作，如果这个过程一直进行到了根节点才停止，则会导致 B 树的高度增加 1

对于 5 阶 B 树，根结点只有达到 5 个关键字时才能产生分裂，成为高度为 2 的 B 树，因此高度为 2 的 5 阶 B 树所含关键字的个数最少是 5

B 树的删除举例：



对于任意一棵包含 n 个关键字、高度为 h 的 m 阶 B 树，满足如下关系：

$$\log_m(n+1) \le h \le \log_{\lceil m/2 \rceil}\left(\frac{n+1}{2}\right) + 1$$

4.求阶乘.py

```
>>> def fac(n):
```

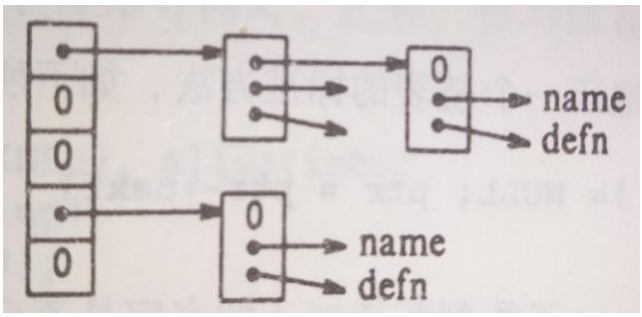
```
    if n==1:                                        #不要漏掉 base case
        return 1
    else:
        return n*fac(n-1)       #Python 代码换页时要用游标卡尺来查看缩进
```

我们来编写一个表查找程序包的核心部分代码，当遇到类似于#define IN 1 之类的程序行时，就需要把名字 IN 和替换文本 1 存入到某个表中，此后当名字出现在某些语句中时就必须替换

```
struct nlist                                        //链表项
{
    struct nlist *next;                             //链表中下一表项
    char *name;                                     //定义的名字
    char *defn;                                     //替换文本
};
#define HASHSIZE 101
static struct nlist *hashtab[HASHSIZE];             //指针表
unsigned hash(char *s)                  //散列函数：为字符串 s 生成散列值
{
    unsigned hashval;
    for(hashval=0;*s!='\0';s++)
        hashval=*s+31*hashval;
    return hashval%HASHSIZE;
};
struct nlist *lookup(char *s)       //lookup 函数：在 hashtab 中查找 s
{
    struct nlist *np;
    for(np=hashtab[hash(s)];np!=NULL;np=np->next)       //遍历链表
        if(strcmp(s,np->name)==0)
            return np;                                  //找到 s
    return NULL;                                        //未找到 s
};
struct nlist *install(char *name,char *defn)
{                           //install 函数：将(name,defn)加入到 hashtab 中
    struct nlist *np;
    unsigned hashval;
    if((np=lookup(name))==NULL)                         //未找到
    {
        np=(struct nlist *) malloc(sizeof(*np));
        if(np==NULL||(np->name=strdup(name))==NULL)
            return NULL;
        hashval=hash(name);
        np->next=hashtab[hashval];
        hashtab[hashval]=np;
    }
    else                                                //已存在
        free((void *)np->defn);                         //释放前一个 defn
    if((np->defn=strdup(defn))==NULL)
        return NULL;
    return np;
};
```

该算法采用的是散列查找方法，将输入的名字转换为一个小的非负整数，该整数随后将作为

一个指针数组的下标。数组中的每个元素指向某个链表的表头，链表中的各个块用于描述具



有该散列值的名字。如果没有名字散列到该值，则数组元素的值为 NULL 散列计算时采用的是无符号算术运算，因此保证了散列值非负，散列函数生成了在数组 hashtab 中执行查找的起始下标，如果该字符串可以被查找到，则它一定位于该起始下标指向的链表的某个块中，具体查找过程由 lookup 函数实现(如果发现表项已存在，则返回指向该表项的指针，否则返回 NULL)

排序问题的定义：

输入：一个 n 个数的**序列**$< a_1, a_2, \ldots, a_n >$

输出：输入序列的一个排列(重排)$< a'_1, a'_2, \ldots, a'_n >$，使得$a'_1 \le a'_2 \le \cdots \le a'_n$

在实际中，待排序的数很少是单独的数值，它们通常是称为**记录**的数据集的一部分，每个记录包含一个关键字，就是排序问题中需要排序的值，记录的剩余部分由**卫星数据**组成，通常是与关键字一起存取的，当一个排序算法重排关键字时，也必须要重排卫星数据。如果每个记录包含大量卫星数据，我们通常重排记录指针的数组，而非记录本身，这样可以降低数据移动量。

集合作为计算机科学的基础，就如同它们在数学中所起的作用，数学中的集合是不变的，而由算法操作的集合却能在过程中增大缩小或发生其他变化，我们称这样的集合是**动态**的，支持插入和删除元素、测试元素是否属于集合这 3 种操作的动态集合称为**字典**。

动态集合的每个元素都由一个对象来表示，如果有一个指向对象的指针，就能对这个对象的各个属性进行检查和操作。某些类型的动态集合假定对象中的一个属性为标识关键字，如果关键字全不相同，可以将动态集合视为一个关键字值的集合。对象可能包含卫星数据，它们与其他对象属性一起移动，除此之外集合的实现不使用它们。

栈和队列都是动态集合，栈实现的是一种**后进先出**策略，被删除的是最近插入的那个元素；而队列实现的是**先进先出**策略，被删除的总是在集合中呆了最长时间的那个元素。

如果试图对一个空栈执行弹出操作，会发生栈下溢(underflow)

45.约瑟夫问题.C

有 n 个小朋友，编号分别为 1,2,….,n，按照编号围成一个圆圈，他们按照顺时针方向从编号为 k 的人由 1 开始报数，报数为 m 的人出列，他的下一个人重新从 1 开始报数，数到 m 的人出列，照这样重复下去，直到所有的人都出列。编写一个算法，输入 n,k,m，按照出列顺序输出编号

可以分 3 步来解决:1.创建一个具有 n 个结点的不带头结点的双向循环链表,编号从 1 到 n，代表 n 个小朋友。2.找到第 k 个结点，即第一个开始报数的人。3.编号为 k 的人从 1 开始报数，并开始计数，报到 m 的人出列即将该结点删除，继续从下一个结点开始报数，直到最后一个结点被删除

```
#include<stdio.h>
#include<malloc.h>
#include<stdlib.h>
typedef int datatype;
typedef struct Node                          //双向链表的类型定义
{
    datatype data;
    struct Node *prior;
    struct Node *next;
}DListNode,*DLinkList;
```

```
void Josephus(DLinkList head,int n,int m,int k)
{       //在长度为 n 的双向循环链表中，从第 k 个人开始报数，数到 m 的人出列
    DListNode *p,*q;
    int i;
    p=head;
    for(i=1;i<k;i++)                                //从第 k 个人开始报数
    {
        q=p;
        p=p->next;
    }
    while(p->next!=p)
    {
        for(i=1;i<m;i++)                                //数到 m 的人出列
        {
            q=p;
            p=p->next;
        }
        q->next=p->next;                    //将 p 指向的结点删除，即出列
        p->next->prior=q;
        printf("%4d",p->data);                      //输出被删除的结点
        free(p);
        p=q->next;                          //p 指向下一个结点，重新开始报数
    }
    printf("%4d\n",p->data);
}
DLinkList CreateDCList(int n)               //创建一个长度为 n 的双向循环链表
{
    DLinkList head=NULL;
    DListNode *s,*q;
    int i;
    for(i=1;i<=n;i++)
    {
        s=(DListNode*)malloc(sizeof(DListNode));
        s->data=i;
        s->next=NULL;
        if(head==NULL)                  //将新生成的结点插入双向循环链表
        {
            head=s;
            s->prior=head;
            s->next=head;
        }
        else
        {
            s->next=q->next;
            q->next=s;
            s->prior=q;
            head->prior=s;
        }
        q=s;                            //q 始终指向链表中的最后一个结点
    }
    return head;
}
int InitDlist(DLinkList *head)                  //初始化双向循环链表
{
    *head=(DLinkList)malloc(sizeof(DListNode));
```

```
    if(!head)
        return -1;
    (*head)->next=*head;
    (*head)->prior=*head;          //
使头结点的 prior 指针和 next 指针指向自己
    return 1;
}
void main()
{
    DLinkList h;
    int n,k,m;
    printf("输入环中人的个数 n=");
    scanf("%d",&n);
    printf("输入开始报数的序号 k=");
    scanf("%d",&k);
    printf("报数为 m 的人出列 m=");
    scanf("%d",&m);
    h=CreateDCList(n);
    Josephus(h,n,m,k);
}
```

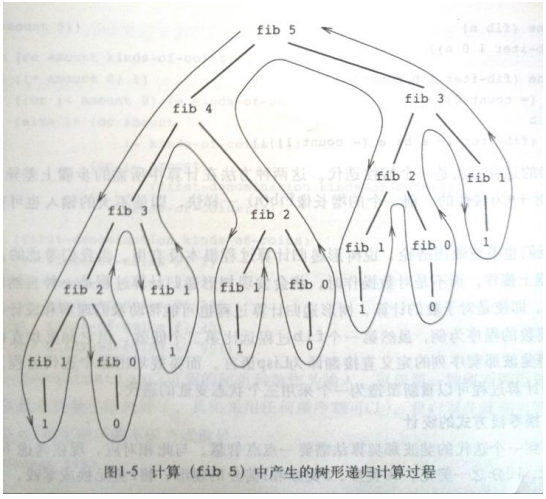


图1-5 计算（fib 5）中产生的树形递归计算过程

缺省(que sheng,default)值就是默认值，是计算机软件要求用户输入初始值而用户未给定时，系统自动赋予的事先设定的数值，野指针指向一个已删除的对象或事先未申请的访问受限的内存区域。避免野指针的方法是初始化时置 NULL 以及释放后置 NULL，因为任何指针变量(除了 static 修饰的)刚被创建时不会自动成为 NULL 指针，它的缺省值是随机的；delete 和 free 只是释放内存空间，并没有把指针干掉，释放之后必须马上把指针赋值为 NULL,可以用这个宏替代 free 操作:#define xfree(x) free(x);x=NULL;

递归计算过程构造起一个推迟进行的操作所形成的链条，收缩阶段表现为这些运算的实际执行。而迭代计算过程就是那种其状态可以用固定数目的状态变量描述的计算过程，而与此同时又存在着一套固定的规则，描述了计算过程在从一个状态到下一个状态转换时，这些变量的更新方式，可能还有一个结束检测，它描述迭代计算过程应该终止的条件。

如果我们在迭代计算过程中的某一步停下来，要想重新唤醒这一计算，只需要提供状态变量的值。而递归计算过程包含了大量的由编译器或解释器本身维持着的“隐含信息”。

我们要分清这 2 个不同的概念：“递归过程”论述的是一个语法形式上的事实，意思是某个过程或某条程序的定义中引用了它本身；“递归计算过程”论述的是这一计算过程在真正执行时的进展。当我们说某个递归过程将产生一个迭代的计算过程时，说明执行计算时只需要跟踪状态变量的轨迹即可。

前面我们写过一个迭代法求解斐波那契数列的程序，其基本想法就是用一对整数 a 和 b，将它们分别初始化为 fib(1)=1 和 fib(0)=0，然后反复地同时使用以下变换(状态转移)规则：

a←a+b ，b←a　　n 次变换后 a 和 b 将分别等于 fib(n+1)和 fib(n)

与线性运行时间的迭代法相比，递归法的效率就低了太多，因为作为树形递归，它做了大量的冗余计算

```
int findmax(int A[],int n){
    int m;
    if(n<=1)
        return A[0];
    else{
        m=findmax(A,n-1);
        return A[n-1]>=m?A[n-1]:m;   }
}                                          //实际效率不如二叉堆
```

上面给了一个递归的例子，findmax 函数返回 n 个数中的最大者

大数据时代，如何获取想要的数据是关键。没有可靠的数据，我们的算法就没有用武之地，也就不存在数据挖掘(挖掘出原始数据中隐藏的有价值的知识)的问题。因此，采集数据是这一切的前提。

最简单的方法是打开 excel，点击“数据”——“自网站”——输入网址——“转到”——点击网页上你想要的区域——“导入”。你就能在 excel 的表格中收集到数据

46.和式分解.C

```
#include<stdio.h>
#define MAX 50
void FenJieHeShi(int HeShu[],int num,int k)
{                           //k:本次(第 k 次)递归调用将要分解出的第 k 个和数
    int j,Same;                                     //j:待分解的数
    for(j=num;j>=1;j--)                             //范围从 1 到 number
    {
        if(j<=HeShu[k-1])
                    //待分解的数小于上一次(第 k-1 次)分解出的和数 HeShu[k-1]
        {                                   //保证分解结果按照不增的顺序排列
            HeShu[k]=j;                 //当前待分解的数存放到序号为 k 的位置
            if(j==num)                                  //已完成，输出之
            {
                printf("%d = %d",HeShu[0],HeShu[1]);
                for(Same=2;Same<=k;Same++)
                    printf("+%d",HeShu[Same]);
                printf("\n");
            }
            else                                //未完成，递归继续下去
                FenJieHeShi(HeShu,num-j,k+1);
        }                                   //还有 num-j 需要进行第 k+1 次分解
    }
}
void main()
{
    int number,HeShu[MAX];
                        //number:准备分解的整数，数组 HeShu:存放分解出来的和数
    printf("please enter a number(<%d)",MAX);
    scanf("%d",&number);
    HeShu[0]=number;                    //初始化第一个和数最大为 number
    printf("\n");
    FenJieHeShi(HeShu,number,1);
}
```

47.上楼梯的走法.C

某人上楼梯一步可以迈 1 个、2 个、或 3 个台阶，假设楼梯有 n 级台阶，求所有可能的上楼梯方法

```
#include<stdio.h>
#define STAIR_NUM 4                                     //楼梯的总台阶级数
int queue[STAIR_NUM];                       //数组 queue 存放每一步上的台阶级数
int total=0;                                        //total:走法数
int index;                                          //index：步数
void output(void)
{
    for(int i=0;i<index;i++)
        printf("_%d",queue[i]);
    printf("\n");
```

```
}
void step(int n)                                    //n 表示还有多少级台阶
{
    if(n==0)                    //n 等于 0 时说明已经构成一个完整的上楼梯方法
    {
        total++;
        printf("————————第%d 种方法————————\n",total);
        output();
        return;
    }
    queue[index++]=1;                           //上 1 级台阶
    step(n-1);                              //还有 n-1 级，递归调用之
    --index;                                //调用结束之后，恢复 index 的值
    if(n>1)                             //n>1 时说明可以上 2 级台阶
    {
        queue[index++]=2;               //同理
        step(n-2);
        --index;
    }
    if(n>2)
    {
        queue[index++]=3;
        step(n-3);
        --index;
    }
}
void main()
{
    printf("%d 级台阶\n",STAIR_NUM);
    printf("    上楼梯的走法：\n");
    step(STAIR_NUM);
    printf("\n 共有%d 种走法\n",total);
}
```

48.汉诺塔问题.C

将 n 个圆盘从柱子 A 原封不动地转移到空的柱子 C 上，移动过程中可以利用空的柱子 B。要求每次只能移动 1 个圆盘，不能改变圆盘堆叠的上下顺序，只能从柱子上方取出堆在最上方的圆盘。

```
#include<stdio.h>
void move(char x,char y){printf("%c-->%c\n",x,y);}
void hanoi(int n,char one,char two,char three)
{                       //函数 hanoi:将 n 个圆盘从柱子 1 借助柱子 2 转移到柱子 3
    if(n==1)                                    //如果只有一个圆盘
        move(one,three);                    //直接转移过去
    else                                    //否则
    {
        hanoi(n-1,one,three,two);
                        //把上面的 n-1 个圆盘从柱子 1 借助柱子 3 转移到柱子 2
        move(one,three);
                        //然后把柱子 1 最下面的那个圆盘直接转移到柱子 3
        hanoi(n-1,two,one,three);
                        //再把 n-1 个圆盘从柱子 2 借助柱子 1 移动到柱子 3
    }
}
```

```
void main()
{
    int n;
    printf("请输入圆盘的个数:");
    scanf("%d",&n);
    printf("移动步骤如下：\n");
    hanoi(n,'A','B','C');
}
```

贪心算法(greedy algorithm)是一种不追求最优解、只需要找到较为满意解的方法，它省去了寻找最优解从而穷举所有可能性所花费的时间，贪心算法分阶段地工作，在每一个决策点可认为所做决定是最好的，而不考虑将来的后果，一般来说，这意味着选择的是某个局部的最优。这种“眼下能够拿到的就拿”的策略即是这类算法名称的来源。当算法终止时，我们希望局部最优就是全局最优。如果是这样的话，那么算法就是正确的，否则算法得到的是一个次最优解。

最明显的贪心算法的例子就是找零钱，我们每一步都在寻找这一步能够找到的最大面值的纸币，这样一来就保证使用最少数目的纸币。

49.找零钱问题.C

```
#include<stdio.h>
#define N 60
int exchange(float n,float *a,int c,float *r)
{
    int m;
    if(n==0.0)                                  //表示已经成功找完
        return 0;                              //返回 0
    if(c==0)                                   //没有合适的找零钱方案
        return -1;
    if(n<*a)
        return exchange(n,a+1,c-1,r);           //继续寻找较小的面值
    else
    {
        *r=*a;                             //将零钱的面额存放到数组 r 中
        m=exchange(n-*a,a,c,r+1);                //继续分解剩下的零钱
        if(m>=0)
            return m+1;                           //返回找零的零钱张数
        return -1;
     }
}
void main()
{
    float rmb[]={100,50,20,10,5,1,0.5,0.1}; //存放人民币的各种面额大小
    int n=sizeof(rmb)/sizeof(rmb[0]),k,i;
    float change,r[N];
    printf("请输入要找的零钱数:");
    scanf("%f",&change);
    for(i=0;i<n;i++)
        if(change>=rmb[i])          //找到第一个小于 change 的人民币面值
            break;
    k=exchange(change,&rmb[i],n-i,r);
//调用 exchange 函数并返回找回零钱的张数
    if(k<=0)                                   //如果返回了小于等于 0 的数
        printf("找不开! \n");                          //则表示找不开零钱
    else                                                   //输出方案
```

```
    {
        printf("找零钱的方案：%.2f=",change);
        if(r[0]>=1.0)
            printf("%.0f",r[0]);
        else
            printf("%.2f",r[0]);
        for(i=1;i<k;i++)
        {
            if(r[i]>=1.0)
                printf("+%.0f",r[i]);
            else
                printf("+%.2f",r[i]);
        }
        printf("\n");
     }
}
```

发现这个程序不能找零一毛钱，只能找 5 毛钱，也就是说存在 bug(逻辑错误)，请你解决它

在许多情况下，**回溯算法**(backtracking)相当于枚举算法的巧妙实现，在某些情形下它相比穷举搜索，工作量有显著的节省。它的一个例子就是在新房子里摆放家具的问题，开始什么也不摆，如果摆到某一步，该步之后所有家具的摆放方式都不理想，那么我们必须撤销这一步并倒退(回溯)到前一步尝试其他方法。这个算法基本上也是暴力穷举式的，但是它并不直接尝试所有的可能。它往往在某一步决策时裁剪掉许多组可能性

50.八皇后问题.C

在一个 8*8 的国际象棋棋盘上放置 8 个皇后，使她们两两之间不能相互攻击，即两两之间都不在同一行、同一列、同一对角线上，求所有可能的摆放方式

```
#include<stdio.h>
#include<stdlib.h>
int m[8][8]={0};                        //表示棋盘的二维数组，初始化为 0
int num=0;                              //可能解的数目
//对于棋盘前 row-1 行已经放置好皇后
//检测在第 row 行、第 column 列放置一枚皇后是否可行
int check(int row,int column)
{                                   //返回值 0 表示 false,1 表示 true
    if(row==1)
        return 1;
    int i,j;                            //纵向只能有一枚皇后
    for(i=0;i<=row-2;i++)
        if(m[i][column-1]==1)
            return 0;
    i=row-2;                            //左上到右下只能有一枚皇后
    j=i-(row-column);
    while(i>=0&&j>=0)
    {
        if(m[i][j]==1)
            return 0;
        i--;
        j--;
    }
    i=row-2;                            //右上到左下只能有一枚皇后
    j=row+column-i-2;
    while(i>=0&&j<=7)
    {
        if(m[i][j]==1)
```

```
            return 0;
        i--;
        j++;
    }
    return 1;
}
void output()                        //当已经放好一个可行解时，输出棋盘
{
    int i,j;
    num++;
    printf("answer %d:\n",num);
    for(i=0;i<8;i++)
    {
        for(j=0;j<8;j++)
            printf("%d ",m[i][j]);
        printf("\n");
    }
}                                    //采用递归函数实现八皇后的回溯算法
//该函数求解当棋盘前 row-1 行已经放置好皇后，在第 row 行放置皇后
void solve(int row)
{
    int j;                               //考虑在第 row 行的各列放置皇后
    for(j=0;j<8;j++)
    {
        m[row-1][j]=1;                        //在其中一列放置皇后
        if(check(row,j+1)==1)                //检测在这一列能不能放皇后
        {
            if(row==8)  //如果可以放，并且这一列是最后一列，则找到了一个解
                output();                                //输出这个可行解
            else                             //如果可以放，但不是最后一列
                solve(row+1);                     //向下一行，继续搜索、求解
        }
        m[row-1][j]=0;
    }                       //取出这一列的皇后，进行回溯，再在其他列放置皇后
}
int main()
{
    solve(1);                                //求解八皇后问题
    return 0;
}
```

压缩存储：为多个值相同的元素只分配一个存储空间，对 0 元素不分配空间，从而节省空间

稀疏矩阵：0 元素占总元素比例很大的矩阵，我们将其中的非 0 元素及其行标、列标构成一个三元组，再按某种规律存储这些三元组。

通常将二维数组、上下三角矩阵、对称矩阵、三对角矩阵存储在一维数组中，有行优先和列优先两种方法，务必注意矩阵和数组的最小下标是 0 还是 1，

对于矩阵和数组最小下标都为 1 的情况，三对角矩阵中元素$A_{i,j}$在数组中位置是 k，则有 K=2i+j-2

求解数独的 Java 程序：

```
public class Sudoku {
    public static void solveSudoku(char[][] board) {
        if(board == null || board.length == 0)
            return;
        solve(board);
```

```
    }
    public static boolean solve(char[][] board){
        for(int i = 0; i < board.length; i++){
            for(int j = 0; j < board[0].length; j++){
                if(board[i][j] == '.'){
                    for(char c = '1'; c <= '9'; c++){
//trial. Try 1 through 9
                        if(isValid(board, i, j, c)){
                            board[i][j] = c;          //Put c for this cell
                            if(solve(board))
                                return true;
//If it's the solution return true
                            else
                                board[i][j] = '.';    //Otherwise go back
                        }
                    }
                    return false;
                }
            }
        }
        return true;
    }
    private static boolean isValid(char[][] board, int row, int col, char
c){
        for(int i = 0; i < 9; i++) {
            if(board[i][col] != '.' && board[i][col] == c) return false;
//check row
            if(board[row][i] != '.' && board[row][i] == c) return false;
//check column
            if(board[3 * (row / 3) + i / 3][ 3 * (col / 3) + i % 3] != '.'
&& board[3 * (row / 3) + i / 3][3 * (col / 3) + i % 3] == c) return false;
//check 3*3 block
        }
        return true;
    }
    public static void main(String[] args){
        char[][] board={{'.','.','5',  '3','.','.',  '.','.','.'},
                        {'8','.','.',  '.','.','.',  '.','2','.'},
                        {'.','7','.',  '.','1','.',  '5','.','.'},
   //此数独号称世界最难数独
                        {'4','.','.',  '.','.','5',  '3','.','.'},
                        {'.','1','.',  '.','7','.',  '.','.','6'},
                        {'.','.','3',  '2','.','.',  '.','8','.'},

                        {'.','6','.',  '5','.','.',  '.','.','9'},
                        {'.','.','4',  '.','.','.',  '.','3','.'},
                        {'.','.','.',  '.','.','9',  '7','.','.'}};
        solveSudoku(board);
        for(int i=0;i<9;i++){
            for(int j=0;j<9;j++)
                System.out.print(board[i][j]);
            System.out.print('\n');
        }
    }
}
```

51.求 N 皇后问题的解的数目.C

```
              q
        (l,r,m,n,i,j)
           {return
           j?i?q(l,
            r,m,n,
         i&i-1,1)+q((
          l|i&-i)*2
           ,(r|i&-
            i)/2,
           m|i&-i,
           n,0,0):
            0:m==
            n?1:q
            (l,r,
            m,n,~
           (l|r|m)
         &n,1);}main
        (n){for(n=0;n
     ++<15;)printf("%d "
    ,q(0,0,0,(1<<n)-1,0,0));}
```

编译这段代码时会出现很多 warning，不用担心，它可以输出正确结果：

1 0 0 2 10 4 40 92 352 724 2680 14200 73712 365596 2279184

Process exited after 5.518 seconds with return value 0

用于查找的数据集合称为查找表，静态查找表支持 2 种操作：查询某个特定的数据元素是否在查找表中、检索满足条件的某个特定的数据元素的各种属性。除此之外还涉及插入与删除操作的查找表称为动态查找表。

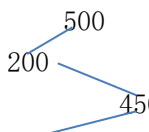
一次查找的长度是指需要比较的关键字次数，而平均查找长度则是所有查找过程中进行比较的关键字的比较次数的平均值。其数学定义为：$\mathrm{ASL}=\sum_{i=1}^{n}P_iC_i$

其中 n 是查找表的长度，P_i是查找第 i 个数据元素的概率，一般认为它们都等于 1/n，C_i是找到第 i 个数据元素所需进行的比较次数。ASL 是衡量查找算法效率的最主要指标。

对于二分(折半)查找算法，表必须满足：有序、以顺序方式存储

在有序单链表或顺序表上做顺序查找的 ASL 相同，都是(n+1)/2。两种查找算法并无绝对快慢之分。

折半查找的判定树是一棵平衡的二叉排序树，查找不成功时和给定值进行关键字的比较次数最多为树的高度，即(向下取整)$(\log_2 n)$+1 或(向上取整)$[\log_2(n+1)]$。



看是否满足二叉排序树的要求，例如 500,200,450,180 不能构成折半查找中关键字比较序列。

折半查找的平均查找长度和最大查找长度都是 O$(\log_2 n)$；二叉排序树的查找性能与数据的输入顺序有关，平均查找长度是 O$(\log_2 n)$，最大查找长度是 O(n)(最坏情况时形成单支树)。

当采用分块查找时，数据的组织方式是：数据分成若干块，每块内数据不必有序，但块间必须有序，每块内最大(或最小)的数据组成索引块

在有 11 个元素的有序表 A[1,2,⋯,11]中进行折半查找：(向下取整)$[\frac{\mathrm{low}+\mathrm{high}}{2}]$，查找元素 A[11]时，被比较的元素下标依次是 6,9,10,11。

哈希表建立了关键字和存储地址之间的一种直接映射关系，散列函数可能会把两个或以上的不同关键字映射到同一地址，这种情况叫做冲突(碰撞)，这些发生碰撞的不同关键字称为

同义词。一个好的散列函数 H(key)满足：1.定义域包含全部关键字，值域的范围依赖于散列表的地址范围，2.散列出来的地址应该尽量在地址空间上均匀分布，3.函数本身的计算过程应该尽量简单。

处理冲突的方法有两种：开放定址法和分离链接法。

开放定址法：可存放新表项的空闲地址既向它的同义词表项开放，也向它的非同义词表项开放，用H_i表示发生冲突后第 i 次探测的散列地址，其递推公式为：$H_i = (H(key) + d_i)\%m$，其中 i=1,2,…,k，m 表示散列表长度，d_i为增量序列，当确定了某一增量序列后，则对应的处理方法是确定的。通常有 3 种取法：线性探测法(d_i=1,2,…,m-1)，再散列法(d_i=$H_2(key)$)，伪随机序列法(d_i =伪随机数序列)，

例如，关键字序列{19,14,23,1,68,20,84,27,55,11,10,79}按散列函数 H(key)=key%13 和线性探测处理冲突得到的散列表是这样的：

1	2	3	4	5	6	7	8	9	10	11	12
14	1	68	27	55	19	20	84	79	23	11	10

给定值 84 的查找过程为：首先 H(84)=6，因哈希表中第 6 项(记作 L6)已经被 19 占领且 19 不等于 84(第 1 次比较)，按线性探测法找到第 1 次处理冲突后的地址(6+1)%16=7，而 L7 也不等于 84，然后 L8=84，查找长度为 3，查找成功

分离链接法：把发生冲突的所有同义词存放在一个线性链表中，这个链表由散列地址唯一标记，我们在前面编写的表查找程序的核心部分代码就是使用了分离链接法

散列表的装填因子=表中记录数 n/散列表长度 m，其平均查找长度直接依赖于其装填因子，由于冲突的出现，使得哈希表的查找依然是一个给定值和关键字比较的过程，尽管哈希表建立了关键字和地址值之间的直接映射使得它能以常数平均时间查找。增大装填因子→冲突可能性增大→降低查找效率。

线性探测法可能会引起聚集(堆积)问题，它只堆积散列表地址项，所以不会影响存储效率，它不改变表中记录数，所以也不影响装填因子，它(果)可能是由不良的散列函数(因)导致的，它会直接导致平均查找长度的增加，使得哈希表丧失了它本来具有的优良的查找性能

满二叉树：高度为 h 且含有$2^h - 1$个结点的二叉树。可以对满二叉树的结点按照从上到下、从左到右的顺序编号，对于编号为 i 的双亲结点，其左右孩子的编号分别是 2i 和 2i+1

完全二叉树：可以看做按照上面的顺序而没有编完结点的满二叉树的一部分，它的叶子只会出现在层次最深的两层上，如果有度为 1 的结点，只可能有 1 个，且一定是一个左孩子

5.二叉树的建立、遍历、销毁.Cpp

```
#include<stdio.h>
#include<malloc.h>
#include<stdlib.h>
#include<iostream>
#include<iomanip>
using namespace std;
typedef char DataType;
#define MAXSIZE 100
typedef struct Node                    //二叉链表的存储结构的类型定义
{
    DataType data;                                      //数据域
    struct Node *lchild;                                //指向左孩子结点
    struct Node *rchild;                                //指向右孩子结点
}*BiTree,BitNode;
/*利用括号嵌套的广义表建立二叉树*/
void CreateBitTree(BiTree *T,char str[])
{
    char ch;
```

```
    BiTree stack[MAXSIZE];            //定义栈，存放二叉树中的结点指针
    int top=-1;                                  //初始化栈顶指针
    int flag,k;
    BitNode *p;
    *T=NULL,k=0;                                //二叉树初始时为空
    ch=str[k];
    while(ch!='\0')                           //如果字符串没有结束
    {
        switch(ch)
        {
            case '(':                     //进入下一层，可能是左孩子
            stack[++top]=p;
            flag=1;
            break;
            case ')':                                //回到上一层
            top--;
            break;
            case ',':                                //是右孩子
            flag=2;
            break;
            default:
            p=(BiTree)malloc(sizeof(BitNode));
            p->data=ch;
            p->lchild=NULL;
            p->rchild=NULL;
            if(*T==NULL)                                 //根结点
                *T=p;                                //创建根结点
            else                                    //已经建立了根结点
            {
                switch(flag)          //根据 flag 的值建立与栈顶结点的关系
                {
                    case 1:
                    stack[top]->lchild=p;
                    break;
                    case 2:
                    stack[top]->rchild=p;
                    break;
                }
            }
        }
        ch=str[++k];                                 //读下一个字符
    }
}
/*按树状打印二叉树*/
void TreePrint(BiTree T,int level)
{
    if(T==NULL)                                      //树为空
        return;                                     //直接返回，什么也不做
    TreePrint(T->rchild,level+1);             //打印右子树，并将层次加 1
    for(int i=0;i<level;i++)                   //按照递归的层次打印空格
        printf("    ");
    printf("%c\n",T->data);                     //输出根结点
    TreePrint(T->lchild,level+1);             //打印左子树，并将层次加 1
}
```

```
/*先序遍历的非递归实现*/
void PreorderTraversal(BiTree T)
{
    BiTree stack[MAXSIZE];                //借助一个栈，存放结点的指针
    int top;                                      //定义栈顶指针
    BitNode *p;                                 //定义一个结点的指针
    top=0;                                        //初始化栈
    p=T;
    while(p!=NULL||top>0)
    {
        while(p!=NULL)            //如果 p 不空，访问根结点，遍历左子树
        {
            cout<<setw(3)<<p->data;                    //访问根结点
            stack[top++]=p;                                  //将 p 入栈
            p=p->lchild;                              //遍历左子树
        }
        if(top>0)                                      //如果栈不空
        {
            p=stack[--top];                           //弹出栈顶元素
            p=p->rchild;                              //遍历右子树
        }
    }
}
/*层次遍历二叉树的算法*/
void LevelOrderTraversal(BiTree T)
{
    BiTree queue[MAXSIZE];             //借助一个队列，用于存放结点的指针
    BitNode *p;
    int front,rear;                     //定义队列的队头指针和队尾指针
    front=rear=-1;                                  //队列初始化为空
    rear++;                                        //队尾指针加 1
    queue[rear]=T;                                 //根结点的指针入队
    while(front!=rear)                             //如果队列不是空的
    {
        front=(front+1)%MAXSIZE;
        p=queue[front];                                //取出队头元素
        cout<<setw(3)<<p->data;                          //输出根结点
        if(p->lchild!=NULL)
        {                       //如果左孩子不是空的，把左孩子的结点指针入队
            rear=(rear+1)%MAXSIZE;
            queue[rear]=p->lchild;
        }
        if(p->rchild!=NULL)
        {                       //如果右孩子不是空的，把右孩子的结点指针入队
            rear=(rear+1)%MAXSIZE;
            queue[rear]=p->rchild;
        }
    }
}
void InorderTraversal(BiTree T)              //中序遍历二叉树的算法
{
    if(T!=NULL)
    {
        InorderTraversal(T->lchild);
```

```
        printf("%2c ",T->data);
        InorderTraversal(T->rchild);
    }
}
void PostorderTraversal(BiTree T)        //后序遍历二叉树的算法
{
    if(T!=NULL)
    {
        PostorderTraversal(T->lchild);
        PostorderTraversal(T->rchild);
        printf("%2c ",T->data);
    }
}
int Depth(BiTree T)                          //计算二叉树的深度
{
    if(T==NULL)
        return 0;
    return
Depth(T->lchild)>Depth(T->rchild)?1+Depth(T->lchild):1+Depth(T->rchild)
;
}
int leaf(BiTree T)                              //求叶子结点的个数
{
    if(!T)                                       //如果二叉树为空
        return 0;                                       //返回 0
    else
        if(!T->lchild&&!T->rchild)        //如果左右子树都为空，返回 1
            return 1;
        else                            //把左右子树的叶子结点个数相加
            return leaf(T->lchild)+leaf(T->rchild);
}
void DestroyBiTree(BiTree *T)                //销毁二叉树的算法
{
    if(*T)                                   //如果二叉树不为空
    {
        if((*T)->lchild)
            DestroyBiTree(&((*T)->lchild));
        if((*T)->rchild)
            DestroyBiTree(&((*T)->rchild));
        free(*T);
        *T=NULL;
    }
}
int main()
{
    BiTree root;
    int len=0;
    char ch,str[MAXSIZE];
    cout<<"输入二叉树的广义表形式:"<<endl;
    while((ch=getchar())!='\n')
    {
        str[len++]=ch;
    }
    CreateBitTree(&root,str);
    cout<<"树状打印二叉树:"<<endl;
```

```
    TreePrint(root,1);
    cout<<"先序遍历："<<endl;
    PreorderTraversal(root);
    cout<<endl;
    cout<<"中序遍历："<<endl;
    InorderTraversal(root);
    cout<<endl;
    cout<<"后序遍历："<<endl;
    PostorderTraversal(root);
    cout<<endl;
    cout<<"层次遍历："<<endl;
    LevelOrderTraversal(root);
    cout<<endl;
    cout<<"这棵二叉树的深度为："<<Depth(root)<<endl;
    cout<<"叶子结点的个数为："<<leaf(root)<<endl;
    DestroyBiTree(&root);
    return 0;}
```

运行结果如下：

```
输入二叉树的广义表形式:
(A(B(C,D),E(F(,G),H(I)))
树状打印二叉树:
            H
                I
        E
                G
            F
    A
            D
        B
            C
先序遍历：
  A  B  C  D  E  F  G  H  I
中序遍历：
 C  B  D  A  F  G  E  I  H
后序遍历：
 C  D  B  G  F  I  H  E  A
层次遍历：
  A  B  E  C  D  F  H  G  I
这棵二叉树的深度为：4
叶子结点的个数为：4
```

52.打印程序本身的程序.C

```
#include<stdio.h>
int main()
{
char *c="#include<stdio.h>%cint main()%c{%cchar
*c=%c%s%c;%cprintf(c,10,10,10,34,c,34,10,10,10);%c}%c";
printf(c,10,10,10,34,c,34,10,10,10);
}
```

```
>>> 3>2
True
>>> '1'<2
False                                          #类型不同，不能比较
>>> aint=7
>>> id(aint)                                 #Python 会给每个变量分一个 ID 号
51864072L
>>> type(aint)                              #获取数据类型
<type 'int'>
>>> 2/3
0
>>> 2.0/3.0
0.6666666666666666
>>> x=5
>>> print y=x+5
SyntaxError: invalid syntax                  #表达式有值，语句没有值
>>> str(9.99)                       #浮点数转为字符串
'9.99'
```

5.生成收敛到 1 的冰雹序列(角谷猜想).py

```
numString=raw_input("Enter a positive integer:")   #默认输入为字符串
num=int(numString)                                  #需要转换为整型
count=0
print "starting with number:",num
while num>1:
    if num%2:                                  #是偶数(odd)
        num=num*3+1
    else:                                       #是奇数(even)
        num=num/2
    print num,",",
    count+=1
else:
    print "sequence is ",count," numbers long"
```

运行结果如下：

```
Enter a positive integer:123
starting with number: 123
370 , 185 , 556 , 278 , 139 , 418 , 209 , 628 , 314 , 157 , 472 , 236 , 118 ,
59 , 178 , 89 , 268 , 134 , 67 , 202 , 101 , 304 , 152 , 76 , 38 , 19 , 58 ,
29 , 88 , 44 , 22 , 11 , 34 , 17 , 52 , 26 , 13 , 40 , 20 , 10 , 5 , 16 ,
8 , 4 , 2 , 1 , sequence is  46  numbers long
>>> print "first line \n second line"       #转义字符：换行
first line
 second line
>>> S="Hello World"
>>> S[10]                             #字符串序列操作
'd'
>>> print S[0],S[7],S[-3]
H o r
>>> S[6:10]                       #获取位置从 6 到 9 的子序列
'Worl'
>>> print S[:5],"  ",S[6:]
Hello    World
>>> print S[::2],"\n",S[::3],"\n",S[::-2],"\n",S[-3::-1]
HloWrd         #从 0 开始，按顺序取得位置 0,2,4,6,8,10 上的字符并合成一个切片
HlWl                                  #0,3,6,9
drWolH                                #负号表示倒序，从-1 开始
```

```
roW olleH                           #字符 r 可以看做 S[8]，也可看做 S[-3]
>>> type(S)
<type 'str'>
>>> print S*3,"\n",(S+' '+"I am a smart AI")*2            #字符串拼接
Hello WorldHello WorldHello World
Hello World I am a smart AIHello World I am a smart AI
>>> 'ghuhu'<'ghvaaa'                #比较字符串的方式与 C/C++的方式相同
True
>>> len(S)                              #获取字符串长度
11
>>> 'Jesus Christ'.find('ris')              #寻找子序列所在位置
8
>>> S.find('z')
-1                            #没有找到
>>> S.upper()                       #转为大写
'HELLO WORLD'
>>> print "%10s is %-10d years old." %("Bill",25)         #多个参数
      Bill is 25        years old.
>>> import math
>>> print "PI is %40.33f"%(math.pi)
PI is      3.1415926535897931159979634685441 85
>>> 'The dark knight'.split(' ')
['The', 'dark', 'knight']
>>> 'The dark knight'.split('k')            #以其中某个字符为界限拆分字符串
['The dar', ' ', 'night']
>>> "abcdefghijklmn"[::-1]                      #倒转字符串
'nmlkjihgfedcba'
>>> print r"Hi,\n today is Friday."             #字母 r 使转义字符失效
Hi,\n today is Friday.
>>> print "the godfather".title()               #标题化
The Godfather
>>> tuple1=('you',456,'English',9.34)       #定义一个元祖
>>> tuple1[2]='France'                      #元祖中元素不能修改
#报错：TypeError: 'tuple' object does not support item assignment
>>> cmp(12,15)                    #比较两者，如果前者小，返回-1
-1
>>> numlist=[2,-4,7,-3,18,9,11]      #定义一个列表
>>> print sorted(numlist)               #列表排序
[-4, -3, 2, 7, 9, 11, 18]
>>> print numlist                    #排序不会修改列表
[2, -4, 7, -3, 18, 9, 11]
>>> numlist[2]=100                   #修改位置 2 上的元素为 100
>>> print numlist
[2, -4, 100, -3, 18, 9, 11]
>>> print sum(numlist)                  #求和
133
>>> print numlist.insert(2,11)              #在位置 2 上插入元素 11
None
>>> numlist.insert(2,11)
>>> print numlist
[2, -4, 11, 11, 100, -3, 18, 9, 11]
>>> print numlist.count(11)             #计算元素 11 的出现次数
3
>>> numlist.append(0)        #在列表末尾添加元素 0
```

```
>>> numlist.pop(0)            #删除位置 0 上的元素
2
>>> print numlist
[-4, 11, 11, 100, -3, 18, 9, 11, 0]
>>> my=[4,6,-1.1,'bob',0,'Alan']
>>> yes=set(my)                      #用列表初始化集合
>>> No=frozenset(['why',0,'what',-1.1,'Alan'])         #不可变集合
>>> print 6 in yes               #元素 6 是否在集合 yes 中
True
>>> print yes>=No                    #集合 No 是不是 yes 的子集
False
>>> print yes-No                 #求差集
set(['bob', 4, 6])
>>> print yes&No                 #求交集
set([0, -1.1, 'Alan'])
>>> No|=yes                   #求并集，同时把结果赋值给 No
>>> print No
frozenset([0, 'what', 4, 6, 'Alan', 'bob', -1.1, 'why'])
>>> No.remove(-1.1)           #删除元素是错误用法，不可变集合中的元素不能变动
AttributeError: 'frozenset' object has no attribute 'remove'
>>> yes.update('sunrise')            #用一个字符串序列来更新集合
>>> print yes
set([0, 4, 6, 'i', 'Alan', 'n', 's', 'r', 'u', 'e', -1.1, 'bob'])
>>> dict1={'name':'leon','country':'France','year':1994}  #定义一个字典
>>> print dict1['name']
leon
>>> print dict1.values()      #返回一个包含字典中所有值的列表
['France', 'leon', 1994]
>>> dict2={'name':'Forrest Gump','actor':'Tom Hanks','year':1994}
>>> dict1.update(dict2)       #合并字典
>>> print dict1
{'name': 'Forrest Gump', 'year': 1994, 'country': 'France', 'actor': 'Tom
Hanks'}
>>> for up in dict2:
        print dict2[up],
Forrest Gump Tom Hanks 1994
>>> print (lambda x,y:x*x+y*y)(3,4)         #定义并直接使用一个匿名函数
25                                   #Python 关键字：lambda
>>> a=9
>>> b=6
>>> a,b=b,a               #交换
>>> print a,b,
6 9
```

6.匿名函数的调用.py

```
import datetime
def name(a):
        return "I will say %s"%a
def call(func,param):
        print datetime.datetime.now()
        print func(param)
if __name__=='__main__':  #这个源文件不用做被其他程序调用的模块而是自己运行
        call(name,'hello!')
        call(lambda x:x*2,9)
```

运行结果如下：

```
2017-08-01 21:03:18.791000
I will say hello!
2017-08-01 21:03:18.807000
18
>>> d={'mi':1,'do':3,'fa':6}
>>> for k,v in d.iteritems():
              print (k,'--->',v)                  #打印字典映射
('do', '--->', 3)
('mi', '--->', 1)
('fa', '--->', 6)
>>> [i**2 for i in xrange(10)]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> colors=['red','blue','pink','green']
>>> for i,color in enumerate(colors):              #列表索引
              print (i,'--->',color),
(0, '--->', 'red') (1, '--->', 'blue') (2, '--->', 'pink') (3, '--->',
'green')
>>> port='Hello Java,Hello Python,Use Javascript!'
>>> print port.replace('Hello','Bye')
Bye Java,Bye Python,Use Javascript!
>>> import re
>>> rex=r'(Hello|Use)'                          #正则表达式
>>> print re.sub(rex,'Bye',port)
Bye Java,Bye Python,Bye Javascript!
>>> shipList=[1,2,3,4,5]
>>> print shipList[10:]                         #没有出现 error
[]
>>> print reduce(lambda x,y : x+y,range(0,10),10)     #累加
55
>>> K=[1,2,3,4,5,6,7,8,9]
>>> print map(lambda i:i+10,filter(lambda i:i%2,K))
[11, 13, 15, 17, 19]            #使用 filter(过滤器)选出奇数并加 10
>>> D={'a':1,'b':2,'c':3,'d':4}
>>> {v:k for k,v in D.items()}                  #反转字典
{1: 'a', 2: 'b', 3: 'c', 4: 'd'}
>>> fun=[1,2,3]
>>> how=['A','B','C']
>>> Z=zip(fun,how)
>>> Z
[(1, 'A'), (2, 'B'), (3, 'C')]
>>> zip(*Z)                                     #转置矩阵
[(1, 2, 3), ('A', 'B', 'C')]
>>> import heapq
>>> heapq.nsmallest(3,[4,7,5,6,10,11,1,7,2,8])     #列表中最小的 3 个数
[1, 2, 4]
>>> heapq.nlargest(3,[-4,7,-10,9,3,-5,1,6,4])
[9, 7, 6]
>>> W=[],
>>> W
([],)                                           #默认是元组
>>> thisG=zip(*([iter(K)]*2))
>>> print thisG
[(1, 2), (3, 4), (5, 6), (7, 8)]
>>> sum(thisG,())
(1, 2, 3, 4, 5, 6, 7, 8)
```

53.打印蛇形方阵.C

```
#include<stdio.h>
#define N 20
void main()
{
    int i,j,a[N][N],n,k;
    printf("请输入矩阵的阶 n= ");
    scanf("%d",&n);
    k=1;
    for(i=1;i<=n;i++)                        //输出上三角(前 n 个斜行)
        for(j=1;j<=i;j++)
        {
            if(i%2==0)
                a[i+1-j][j]=k;
            else
                a[j][i+1-j]=k;
            k++;
        }
    for(i=n+1;i<2*n;i++)                     //输出下三角(后 n-1 个斜行)
        for(j=1;j<=2*n-i;j++)
        {
            if(i%2==0)
                a[n+1-j][i-n+j]=k;
            else
                a[i-n+j][n+1-j]=k;
            k++;
        }
    printf("********蛇形方阵*********\n");
    for(i=1;i<=n;i++)
    {
        for(j=1;j<=n;j++)
            printf("%4d",a[i][j]);
        printf("\n");
    }
}
```

运行结果如下：

```
请输入矩阵的阶 n= 5
********蛇形方阵*********
   1   3   4  10  11
   2   5   9  12  19
   6   8  13  18  20
   7  14  17  21  24
  15  16  22  23  25
```

6.高精度整数的加法.Cpp

```
#include<stdio.h>
#include<string.h>
struct bigInteger   //高精度整数结构体
{
    int digit[1000];              //按 4 位数一个单位保存保存数值
    int size;                     //下一个我们未使用的数组单元
    void init()            //C++特有的构造函数，作用是初始化结构体
    {
        for(int i=0;i<1000;i++)
```

```
            digit[i]=0;                         //所有数位清零
        size=0;                         //没有一个数组单元被使用
    }
    void set(char str[])                        //从字符串中提取整数
    {
        init();                                     //对结构体初始化
        int L=strlen(str);                        //计算字符串长度
        for(int i=L-1,j=0,t=0,c=1;i>=0;i--)
//从最后一个字符开始倒序遍历字符串
        {
//j 控制每 4 个字符转换为 1 个数字存入数组，t 临时保存字符转换为数字的中间值
//c 表示当前位的权重，按 1,10,100,1000 顺序变化
            t+=(str[i]-'0')*c;
//计算这个四位数中当前字符代表的数字，即数字乘以当前位权重
            j++;                                    //当前转换字符数增加
            c*=10;                                     //计算下一位权重
            if(j==4||i==0)
    //若已经连续转换 4 个字符，或者已经到达最后一个字符
            {   //将这 4 个字符代表的 4 位数存入数组，size 移动到下一个数组单位
                digit[size++]=t;
                j=0;                        //重新开始计算下 4 个字符
                t=0;                              //临时变量清零
                c=1;                              //权重变为 1
            }
        }
    }
    void output()                                 //输出该高精度整数
    {
        for(int i=size-1;i>=0;i--)
        {
            if(i!=size-1)                     //当前数字不足 4 位时用 0 补充
                printf("%04d",digit[i]);
            else                        //如果是最高位，则不需要输出前导的 0
                printf("%d",digit[i]);
        }
        printf("\n");
    }
    bigInteger operator + (const bigInteger &A) const   //重载加法运算符
    {
        bigInteger ret;                         //返回值，即两数相加的结果
        ret.init();                            //对其初始化
        int carry=0;                          //进位，初始值为 0
        for(int i=0;i<A.size||i<size;i++)
        {                           //计算两个整数当前位以及来自低位的进位和
            int tmp=A.digit[i]+digit[i]+carry;
            carry=tmp/10000;                      //计算该位的进位
            tmp%=10000;                           //去除进位部分，取后 4 位
            ret.digit[ret.size++]=tmp;            //保存该位结果
        }
        if(size!=0)                           //计算结束后如果最高位有进位
            ret.digit[ret.size++]=carry;          //保存该进位
        return ret;                                 //返回
    }
```

```
}a,b,c;
char str1[1002],str2[1002];
int main()
{
    while(scanf("%s%s",str1,str2)!=EOF)          //输入
    {                                   //用两个字符串分别设置两个高精度整数
        a.set(str1);b.set(str2);
        c=a+b;                           //计算它们的和
        c.output();                             //输出
    }
    return 0;
}
```

上面的代码实际上定义了一个高精度运算的类 bigInteger，在 C++中，不需要 typedef 就可以直接用结构体名来定义构造函数，并且还提供自动初始化的功能。operator 是一个 C++的关键字，可以用来自己重新定义运算符，如果是加法运算符，圆括号中需要两个参数，但是现在是把重载运算符放在了结构体内部，所以只需要一个参数&A 和一个返回值 ret，同时要注意参数和返回值属于哪个类(这里属于 bigInteger 类)

54.高精度阶乘.C

```
#include<stdio.h>
#include<string.h>
#include<time.h>
const int MAXN=3000;
int main()
{
    int f[MAXN];
    int i,j,n;
    scanf("%d",&n);
    memset(f,0,sizeof(f));           //对数组 f 中每一个元素赋值 0
    f[0]=1;
    for(i=2;i<=n;i++)
    {                          //乘以 i
        int c=0;
        for(j=0;j<=MAXN;j++)
        {
            int s=f[j]*i+c;
            f[j]=s%10;
            c=s/10;
        }
    }
    for(j=MAXN-1;j>=0;j--)
        if(f[j])
            break;             //忽略前导 0
    for(i=j;i>=0;i--)
        printf("%d",f[i]);
    printf("\n");
    printf("Time used = %.21f\n",(double)clock()/CLOCKS_PER_SEC);
//打印程序运行的时间
    return 0;
}
```

55.快速排序算法.C

```
#include<stdio.h>
/*对数组 a[low...high]的元素进行一趟排序，使枢纽前面的元素小于枢纽元素，枢纽后面的元素大于等于枢纽元素，并返回枢纽位置*/
```

```
int Partition(int a[],int low,int high)
{
    int t,pivot;
    pivot=a[low];                    //将表的第一个元素作为枢纽元
    t=a[low];
    while(low<high)                   //从表的两端交替地向中间扫描
    {
        while(low<high&&a[high]>=pivot)         //从末端向前扫描
            high--;
        if(low<high)
        {                       //将当前 high 指向的元素保存在 low 位置
            a[low]=a[high];
            low++;
        }
        while(low<high&&a[low]<=pivot)          //从前向后扫描
            low++;
        if(low<high)
        {                  //将当前 low 指向的元素保存在 high 位置
            a[high]=a[low];
            high--;
        }
        a[low]=t;                  //将枢纽元保存在 low 位置
    }
    return low;                                //返回枢纽所在位置
};
void DispArray(int a[],int n)           //打印数组
{
    int i;
    for(i=0;i<n;i++)
        printf("%4d",a[i]);
    printf("\n");
};
void divide(int a[],int n,int pivot,int count)
{                                           //打印每次划分的结果
    int i;
    printf("第%d 次划分结果：[",count);
    for(i=0;i<pivot;i++)
        printf("%-4d",a[i]);
    printf("]");
    printf("%3d ",a[pivot]);
    printf("[");
    for(i=pivot+1;i<n;i++)
        printf("%-4d",a[i]);
    printf("]");
    printf("\n");
};
void Qsort(int a[],int n,int low,int high)      //快速排序
{
    int pivot;
    static int count=1;
    if(low<high)                      //如果元素序列的长度大于 1
    {
        pivot=Partition(a,low,high);          //划分
        divide(a,n,pivot,count);              //打印本次划分结果
        count++;
```

```
        Qsort(a,n,low,pivot-1);            //对左边的子集快速排序
        Qsort(a,n,pivot+1,high);           //对右边的子集快速排序
    }
};
//驱动快速排序算法
void QuickSort(int a[],int n){Qsort(a,n,0,n-1);};
int main(){
    int a[]={37,19,43,22,22,89,26,92};
    int n=sizeof(a)/sizeof(a[0]);
    printf("快速排序前：");
    DispArray(a,n);
    QuickSort(a,n);
    printf("快速排序结果：");
    DispArray(a,n);
    return 0;}
```

算法运行结果如下：

快速排序前： 37 19 43 22 22 89 26 92

第 1 次划分结果：[26 19 22 22] 37 [89 43 92]

第 2 次划分结果：[22 19 22] 26 [37 89 43 92]

第 3 次划分结果：[19] 22 [22 26 37 89 43 92]

第 4 次划分结果：[19 22 22 26 37 43] 89 [92]

快速排序结果： 19 22 22 26 37 43 89 92

可见快速排序是一种基于分治与递归思想的排序算法

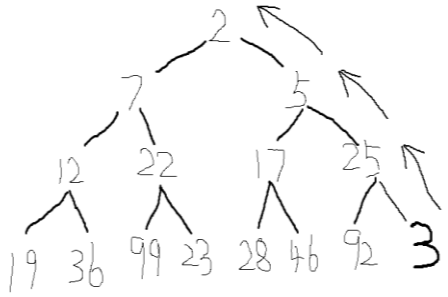
堆的定义如下：n 个关键字序列 L[1…n]是一个堆，当且仅当该序列满足以下两种情况之一：

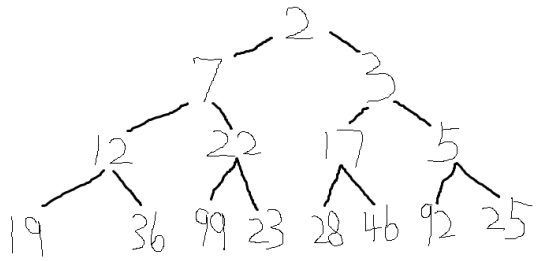
大根(顶)堆：L(i)>=L(2i)且 L(i)>=L(2i+1)

小根(顶)堆：L(i)<=L(2i)且 L(i)<=L(2i+1)　　条件：1<=i<=(向下取整)(n/2)

可见堆是一棵特殊的完全二叉树。**优先队列**是允许至少这样两种操作的数据结构：插入、删除最小者。如果使用普通的线性的队列来实现这两种功能，寻找最大(小)者需要枚举整个队列，时间复杂度高，插入又需要移动很多元素，空间复杂度高。而堆就是优先队列的一种有效的实现，插入操作的时间复杂度可以从 O(N)降低到 O(logN)

这是一个小根堆，每一个爹结点都不比它的两个儿子结点大。堆对应的数组的下标是这棵二





叉树的层次遍历序列，如果想插入一个新元素，就把它放在堆的末尾位置，然后不断地和它的爹结点比较，如果比爹结点小，说明不满足小根堆的要求，需要把它和它的爹结点交换，直到满足小根堆特性为止。这种过程叫做向上调整。

创建堆的算法是这样的：有一个 n 个元素的序列，把这 n 个结点以自顶向下、从左到右的方式从 1 到 n 编码，这样就把该序列变成了一棵完全二叉树，然后从最后一个结点开始，依次判定以这个结点为根的子树是不是一个小根堆，如果所有子树都是小根堆，那么整个树也就是一个建成了的堆；如果发现了不符合的子树，就要向下调整，不管是向上还是向下，都是把爹结点和孩子结点交换，使它满足顺序。事实上叶子结点没有子树，所以略过它们，直接

从最后一个非叶结点(编号为 n/2)开始判定

堆可以实现排序算法，我们根据待排序的序列建一个大根堆，建好之后，最大的元素在 h[1]，因为我们要从小到大排序，希望最大的放在最后，所以要把 h[1]和 h[n]交换，然后向下调整 h[1]以保持堆的特性，OK，现在最大的元素已经归位，它不再属于堆，所以要把堆的大小减一即 n--，然后反复直到堆的大小变成 1 为止，此时已经归位的数组就是排好序的了

56.堆排序算法.C

```
#include<stdio.h>
int h[101];         //用来存放堆的数组
int n;              //堆中元素个数，即堆的大小
void swap(int x,int y)      //交换堆中两个元素的值
{
    int t=h[x];
    h[x]=h[y];
    h[y]=t;
}
//向下调整函数
void siftdown(int i)
/*传入一个需要向下调整的结点编号 i，这里传入 1，表示从堆的顶点开始向下调整*/
{
    int t,flag=0;                   //flag 用来标记是否需要继续向下调整
    while(i*2<=n&&flag==0)
        //当 i 结点至少有左儿子并且需要继续调整的时候循环就执行
    {
        if(h[i]<h[i*2])             //首先判断它和左儿子的关系
            t=i*2;
        else                         //并用 t 来记录较大的结点编号
            t=i;
        if(i*2+1<=n)                 //如果它有右儿子，再对右儿子讨论
        {
            if(h[t]<h[i*2+1])           //如果右儿子的值更大
                t=i*2+1;                //更新较小的结点编号
        }
        if(t!=i)    //如果发现最大的结点编号不是自己，说明子结点中有比爹更大的
        {
            swap(t,i);                  //交换它们
            i=t;
            //把 i 更新成刚才与它交换的子结点的编号，便于接下来继续向下调整
        }
        else
            flag=1;                 //否则说明爹结点比子结点都大，不用继续了
    }
}
void creat()            //建堆
{
    for(int i=n/2;i>=1;i--)
        siftdown(i);
}
void heapsort()             //堆排序
{
    while(n>1)
    {
        swap(1,n);
        n--;
```

```
        siftdown(1);
    }
}
int main()
{
    int i,num;
    scanf("%d",&num);
    for(i=1;i<=num;i++)
        scanf("%d",&h[i]);
    n=num;
    creat();
    heapsort();
    for(i=1;i<=num;i++)
        printf("%d ",h[i]);
    return 0;
}
```

二路归并排序的主要思想是：把数组中每个元素看做一个有序的子序列，然后不断地把相邻的两个子序列合并，合并之后的子序列依然要保持有序。直到有序子序列个数为 1，整个表就完成了排序

57.归并排序算法.C

```
#include<stdio.h>
#include<stdlib.h>
void CopyArray(int source[],int dest[],int len,int start)
{       //把源数组中的元素复制到目标数组中
    int i,j=start;      //len 是源数组长度，start 是目标数组起始位置
    for(i=0;i<len;i++)
    {
        dest[j]=source[i];
        j++;
    }
}
void DispArray(int a[],int n)   {
    for(int i=0;i<n;i++)
        printf("%4d",a[i]);
    printf("\n");};
void Merge(int a[],int left,int right)  //合并两个子序列中的元素
{
    int begin1,begin2,mid,k=0,len,*b;
    begin1=left;
    mid=(left+right)/2;
    begin2=mid+1;
    len=right-left+1;
    b=(int *)malloc(len*sizeof(int));
    while(begin1<=mid&&begin2<=right)
    {
        if(a[begin1]<=a[begin2])
            b[k++]=a[begin1++];
        else
            b[k++]=a[begin2++];
    }
    while(begin1<=mid)
        b[k++]=a[begin1++];
    while(begin2<=right)
        b[k++]=a[begin2++];
    CopyArray(b,a,len,left);
```

```
    free(b);
}
void MergeSort(int a[],int left,int right)         //归并排序
{
    int i;
    if(left<right)
    {
        i=(left+right)/2;
        MergeSort(a,left,i);
        MergeSort(a,i+1,right);
        Merge(a,left,right);
    }
}
int main(){
    int a[]={6,5,7,4,8,3,9,2,0,10,-1,17,-8};
    int len=sizeof(a)/sizeof(a[0]);
    MergeSort(a,0,len-1);
    DispArray(a,len);
    return 0;
}
```

决策树是一棵二叉树，每个节点表示在元素之间的所有可能的一组排序，它与已经进行的比较一致，比较的过程是树的边，之所以是二叉树，是因为每次分叉只能比较其中两个元素

位向量是一个仅包含 0 和 1 的数组。

58.冒泡排序算法.C

```
#include<stdio.h>
int main()
{
    int a[100],i,j,n;
    scanf("%d",&n);
    for(i=1;i<=n;i++)
        scanf("%d",&a[i]);
//冒泡排序算法的核心部分:
    for(i=1;i<=n-1;i++)                        //n 个数排序，只用进行 n-1 趟
        for(j=1;j<=n-i;j++)        //从第 1 位开始比较直到最后一个尚未归位的数
            if(a[j]>a[j+1])                                //比较相邻元素大小
                a[j]^=a[j+1]^=a[j]^=a[j+1];                //交换
//算法的代码最短可以压缩到 4 行
    for(i=1;i<=n;i++)
        printf("%5d",a[i]);
    getchar();
    return 0;
}
```

冒泡排序的基本思想是：每次比较两个相邻的元素，如果它们的顺序错误就把它们交换过来这样一直比较下去直到最后 2 个数比较完毕后，最小的数就归位了，就如同一个气泡一步一步往后翻滚。每把一个数归位(放到了最终位置上)我们就说完成了一趟排序。

排序算法的稳定性是指经过排序后，能使关键字相同的元素保持原来序列中的相对位置不变，稳定性不能衡量一个排序算法的优劣。在排序过程中根据数据元素是否始终在内存中可以把排序算法分为内部排序和外部排序。

在链表上也可以实现排序算法，但并非所有能在顺序表上实现的排序都能在链表上实现，例如折半插入排序对于链表而言不适用。

最简单的排序算法之一是插入排序。它由 N-1 趟排序组成，对于 P=1 到 P=N-1 趟，保证从位置 0 到位置 P 上的元素为已排序的状态，在第 P 趟，将位置 P 上的元素向左移动并插入到它

在前 P+1 个元素中的正确位置上。

```
typedef int ElementType;
void InsertionSort(ElementType A[],int N)
{
    int j,P;
    ElementType tmp;
    for(P=1;P<N;P++)
    {
        tmp=A[P];
        for(j=P;j>0&&A[j-1]>tmp;j--)
            A[j]=A[j-1];
        A[j]=tmp;
    }
}
```

希尔排序算法通过比较相距一定间隔的元素来工作，各趟比较所用的间隔的距离随着算法的进行而减小，直到只比较相邻元素的最后一趟为止，因此它也被称为缩小增量排序。

希尔排序使用一个增量序列$h_1, h_2, \ldots, h_t$，只要$h_1 = 1$，任何增量序列都是可行的，不过有些增量序列比另外一些增量序列更好。在使用增量h_k的一趟排序之后，对于每一个 i 我们有$A[i] \le A[i + h_k]$，所有相隔h_k的元素都被排序，此时称表是h_k-排序的

对于增量h_k，表中记录被分成h_k个子表，每个子表中的每个元素在原来的表中相隔h_k，所有子表构成一个组，然后对组内每一个子表执行直接插入排序。重复这个过程，直到最后一组，最后一组使用的增量为 1。

初始	81	94	11	96	12	35	17	95	28	58	41	75	15
5-排序后	35	17	11	28	12	41	75	15	96	58	81	94	95

插入排序、归并排序不能保证每一趟排序后把一个元素归位，插入排序甚至还可能出现最后一趟开始之前所有元素都没归位的情况

确定当前记录在左边有序子表的插入位置时，直接插入排序采用顺序查找，而折半插入排序采用折半查找。两者的不同之处仅仅在于元素之间的比较次数。

快速排序和冒泡排序一样，是基于元素之间的交换的，具体操作是这样:选一个枢纽，先从右到左扫描比枢纽小的元素并和枢纽交换，再从左到右扫描比枢纽大的元素并和枢纽交换，再不断重复这种操作，直到枢纽归位就完成了一趟排序。这些细节在编程时被隐藏在了递归当中。

对于快速排序，当每趟的枢纽都把表等分成两个长度相近的子表时，速度最快；当表本身基本有序或逆序时，速度最慢。

选择排序每次从未排序的记录中选择最小关键字的记录加入到已排序记录的末尾，事实上堆排序也是基于这样的思想

删除堆顶元素时，先把堆的最后一个元素和堆顶元素交换，由于此时堆的性质被破坏，需要对此时的根结点(堆顶元素)执行向下调整操作；执行插入操作时，先把要插入的结点放在堆的末尾，然后对这个结点执行向上调整操作。

冒泡排序的趟数与序列的初始状态有关；选择排序、归并排序的比较次数与序列的初始状态无关。

把两个各有 N 个元素的有序表合并成一个有序表，最少和最多的比较次数分别是 N 和 2N-1

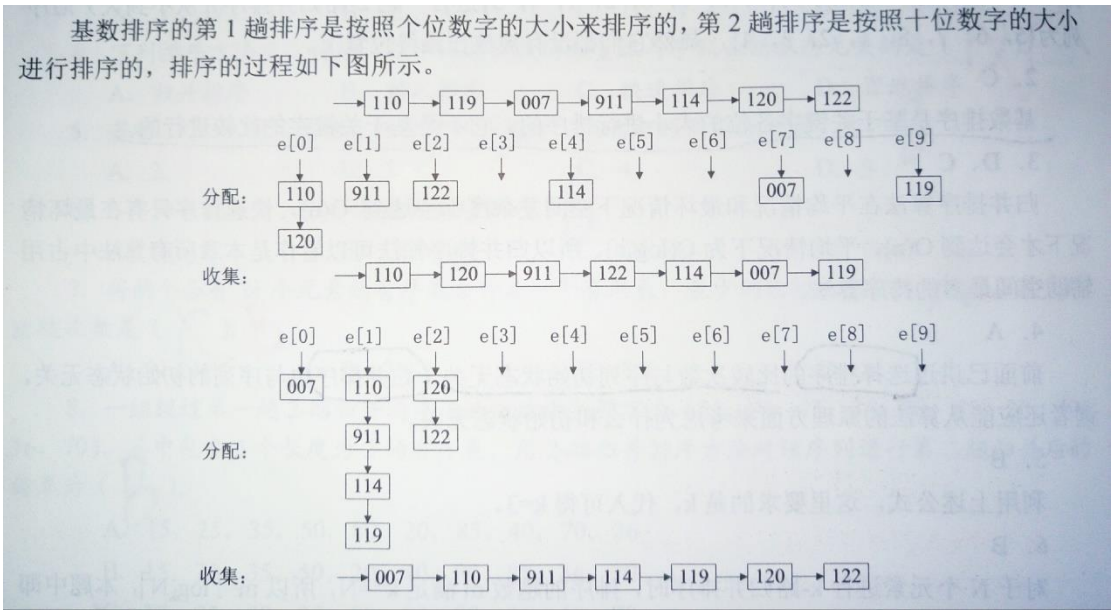
基(计)数排序是一种很特别的排序算法，它不直接比较关键字而是比较关键字各位的大小，借助分配和收集两种操作

59.房屋窃贼.C

你是一个专业的强盗并且计划抢劫沿街的房屋。 每个房子都有一定的钱藏在里面，阻止你抢劫的唯一因素是如果两个相邻的房子在同一个晚上被闯入，它会自动联系警方。给出一个

代表每个房子的金额的非负整数数组，确定你可以抢到的最高金额。
对每一个房屋有两种可能：抢或者不抢，这个房屋后面的房屋同样也有这两种可能，这两种情况获得的金钱可以由前面房屋的两种可能性推导出来，一个简单的循环就能完成

```
#define max(a,b) ((a>b)?a:b)
int rob(int* nums, int numsSize) {
    int not=0;
    int yes=0;
    for(int i=0;i<numsSize;i++){
        int tmp=yes;
        yes=not+nums[i];
        not=max(not,tmp);
    }
    return max(not,yes);
}
```



给你一个整型数组 nums，要你返回一个新的 res 数组。 res 数组的值是 res [i]是 nums 数组中 nums [i]右侧的比 nums[i]小的元素的数量，(Java 实现)

```
    public List<Integer> countSmaller(int[] nums) {
        List<Integer> res = new LinkedList<>();
        List<Integer> sorted = new ArrayList<>();
        for(int i = nums.length - 1; i >= 0; i--){
            int index = findless(sorted, nums[i]);
            res.add(0, index);
            sorted.add(index, nums[i]);
        }
        return res;
    }
    public int findless(List<Integer> sorted, int target){
        if(sorted.isEmpty())  return 0;
        int left = 0, right = sorted.size();
        while(left < right){
            int mid = (left + right) / 2;
            if(sorted.get(mid) >= target)    right = mid;
            else   left = mid + 1;
        }
        return left;
    }
```

<table>
<tr><td rowspan="2">排序算法的种类</td><td colspan="3">时间复杂度</td><td rowspan="2">空间复杂度</td><td rowspan="2">是否稳定</td></tr>
<tr><td>最好</td><td>平均</td><td>最坏</td></tr>
<tr><td>插入排序</td><td rowspan="2">O(N)</td><td colspan="2" rowspan="3">O(N^2)</td><td rowspan="4">O(1)</td><td rowspan="2">是</td></tr>
<tr><td>冒泡排序</td></tr>
<tr><td>选择排序</td><td>O(N^2)</td><td rowspan="5">否</td></tr>
<tr><td>希尔排序</td><td colspan="3"></td></tr>
<tr><td>快速排序</td><td colspan="2" rowspan="3">O(NlogN)</td><td>O(N^2)</td><td>O(logN)</td></tr>
<tr><td>堆排序</td><td>O(NlogN)</td><td>O(1)</td></tr>
<tr><td>2路归并排序</td><td rowspan="2">O(N)</td><td rowspan="2">是</td></tr>
<tr><td>基数排序</td><td colspan="4"></td></tr>
</table>

接下来复习一下数据结构的基本概念：

顺序表是一种支持随机存取(通过首地址和元素序号在O(1)时间内访问)的顺序存储结构，顺序存取是一种读写方式，不是存储方式。

在一个长度为n的顺序表中删除第i个元素时需要把序号为i+1到n的全体元素向左移动一位，这里1<=i<=n；如果是在第i个位置插入一个元素，i的合法取值范围是1<=i<=n+1

静态链表是使用数组实现的链式存储结构，结点的指针域指向下一个结点的相对地址(数组下标，这种方法叫做游标实现)，静态链表需要分配较大的连续空间，插入和删除不需要移动元素。

对于一个线性表的单链表实现和顺序表实现，删除所有值为某个x的元素的操作的时间复杂度相同，空间复杂度不同。顺序输出前k个元素的操作则没有区别

单链表设置头结点的目的是方便运算的实现。插入和删除的算法统一了，不再需要判断是否在第1个元素之前插入或删除第1个元素；空表和非空表的处理统一了，无论链表是否为空，其头指针指向头结点并且保持不变。

删除单链表的最后一个结点需要从表头遍历到表尾，然后把末尾结点的前驱结点的指针改成NULL，需要的时间是O(N)，与表长有关

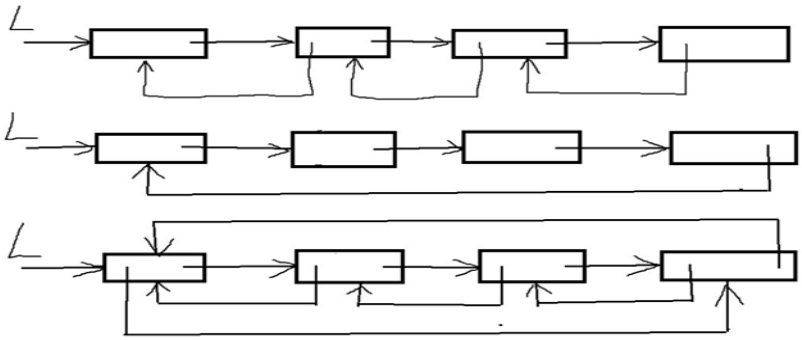
双链表能很方便地访问前驱后继，所以插入和删除操作很方便。在链表的末尾插入和删除一个结点时，需要修改其相邻结点的指针域，而寻找尾结点以及尾结点的前驱操作，只有带头结点的双循环链表用时最短。

双循环链表判断为空的条件是头结点的prior(前驱指针)和next(后继指针)都指向它自己

在双链表中向p结点之前插入一个结点q的操作为

```
p->prior->next=q;  q->next=p;  q->prior=p->prior;   p->prior=q;
```

下面分别展示了双链表、单循环链表、双循环链表



N个不同元素组成的序列进入初始为空的栈，若元素进栈后可停留可出栈，则可能的出栈序列的总个数为$\frac{1}{N+1}C_{2N}^{N}$，这个公式叫做卡特兰(Catalan)数

采用共享栈的好处是节省存储空间，降低上溢的可能性

对于循环队列，当队头指针 Q.front=Maxsize-1 后，再前进一个位置就自动跳转到 0，这种操作可以用取余运算(%或 mod)来实现。为了对循环队列区分队空和队满的情况，入队时少用一个队列单元，并约定以 front 指针在 rear 指针的下一位置作为队满的标志

队满条件：(Q.rear+1) % Maxsize == Q.front

队空条件：Q.front==Q.rear

队列中元素的个数：(Q.rear-Q.front+Maxsize)%Maxsize

双端队列(deque)是指允许两端都可以进行入队和出队操作的队列，它分为输出受限和输入受限，这里的输入输出其实就是入队出队的意思，但操作可能在其中一端被禁止。既然如此，输出受限的双端队列有时就像一个栈。

消除递归不一定要使用栈，也可以用迭代。栈通常用于处理函数或过程调用、存储函数的局部变量。队列可以用于缓冲区，图的广度优先搜索类似于树的层次遍历，也需要借助队列来实现

假设循环队列存储在一维数组 A[0..n-1]中，且队列非空时 front 和 rear 分别指向队头元素和队尾元素，若初始时队列为空，且要求第 1 个进入队列的元素存储在 A[0]处，那么当第 1 个元素入队并放在 A[0]时，front 和 rear 值都是 0，因为队中只有一个元素，所以队头元素就是队尾元素。因为入队时要执行(rear+1)%n 操作(出队也是一样的操作，只是指针改成 front)，所以如果入队后 rear 指向 0，说明 rear 初始值为 n-1，而由于第 1 个元素在 A[0]，入队只改变 rear 指针，所以说明 front 初始值就是 0，没有变化

如果用链表实现队列，最好用带队头指针和队尾指针的非循环单链表，最坏的情况是用只带队头指针的非循环双链表。因为队列需要在两端操作，如果使用循环链表，在入队和出队后还要修改为循环的，这对于队列而言是多余的，但是循环一般能保证快速查找到队头和队尾

完全二叉树和满二叉树均可采用顺序存储结构。

对于只有度为 0 和度为 2 的结点的二叉树，当除根结点只有一个结点外，其他层都有 2 个结点时，结点数最少，深度最深。

事实上二叉树就是一个含有 n 个结点的二叉链表，其中包含 n+1 个空指针

先序序列和后序序列不能唯一确定一棵二叉树，但能确定二叉树中结点的祖先关系，先序序列的首元素和后序序列的末尾元素一定是根结点，先序序列的第 2 个元素和后序序列的倒数第 2 个元素一定是根结点的左孩子。

先序序列和中序序列的关系相当于把先序序列作为入栈次序，中序序列就是可能的出栈次序。

二叉树的遍历实际上就是对非线性结构进行线性化，使得每一个结点都有一个直接的前驱或后继，对二叉树的**线索化**，实际上就是遍历一次二叉树，只是在遍历时检查当前结点的左右指针是否为空，若为空，把它们指向前驱或后继(改为线索)。线索二叉树是一种存储结构

在二叉树线索化时，通常规定：若无左子树，令 lchild 指向其前驱结点；若无右子树，令 rchild 指向其后继结点。如图 4-9 所示，还需要增加两个标志域表明当前指针域所指对象是指向左（右）子结点还是直接前驱（后继）。

ltag	lchild	data	rchild	rtag

图 4-9　线索二叉树的结点结构

其中标志域含义如下：

$$ltag\begin{cases}0 & \text{lchild域指示结点的左孩子}\\ 1 & \text{lchild域指示结点的前驱}\end{cases}$$

$$rtag\begin{cases}0 & \text{rchild域指示结点的右孩子}\\ 1 & \text{rchild域指示结点的后继}\end{cases}$$

后序线索树的遍历仍然需要栈的支持，并且仍然不能有效求解后序后继。这是因为如果该结点有右孩子和双亲，后序序列中它的后继是双亲，右指针无法指向后继。
树转二叉树的画法：在各个兄弟结点之间加一连线；对每一个结点，只保留它与第一个子结点的连线，与其他子结点的连线全部抹掉(左孩子右兄弟)；以树根为轴心，顺时针旋转 45°
森林转二叉树的画法：将每一棵树的根相连；把森林中的每一棵树转为二叉树；以第一棵树的根为轴心顺时针旋转 45°
把树转为二叉树时，树中每一个非叶结点的所有子结点中的最右子结点和根结点在转换后没有右孩子，所以对应的二叉树中无右孩子的结点个数=原来的树中非叶结点数+1
把森林转为二叉树时，森林中的叶结点就是二叉树中左孩子指针为空的结点
树的后根遍历相当于对应二叉树的中序遍历
树中结点往往被附带一个特殊值，称为该结点的权，从树的根结点到任意结点的路径长度(经过的边数)*该结点权值=该结点的带权路径长度。树中所有叶子结点的带权路径长度之和=树的带权路径长度(WPL)。在所有的包含 N 个带权叶子结点的二叉树中 WPL 值最小的那个称为**哈夫曼树**。
如果没有一个编码是另一个编码的前缀，那么这些编码组成一组**前缀编码**。可变长度编码比固定长度编码好很多，因为它对出现频率高的字符赋以较短的编码，而对出现频率低的字符赋以较长的编码，从而压缩了数据量。**哈夫曼编码**是一种有效、被广泛应用的、总长度最短的二进制前缀编码。
生成哈夫曼编码的方法是：首先把每个出现的字符作为一个结点，把它的出现频率作为它的权值，构造出对应的哈夫曼树，于是所有的字符结点都是叶子结点，然后把从根到这个结点的路径上的边做好标记，转向左孩子的标 0，转向右孩子的标 1，这样一来每个字符就有了一个对应的二进制编码
构造哈夫曼树的方法实际上是基于贪心算法的：给定 N 个权值分别为$w_1, w_2, \ldots, w_N$的结点，把这些结点各自作为一棵二叉树，组成一个森林；在森林外面新建一个结点，从森林中拿出两个根结点权值最小的二叉树作为新结点的左右子树，新结点的权等于左右两棵子树的两个根结点权的和；把这棵新建的二叉树放回森林；重复以上操作，直到森林中只剩下一棵树为止，这棵树就是构造成功的哈夫曼树。
可以看出哈夫曼树的一些特点：叶子结点数=(总结点数+1)/2；不存在度为 1 的结点
C++在 C 的基础上嫁接了面向对象，现在我们来看看这部分内容
面向对象编程的特点是继承和动态绑定，C++通过类的派生支持继承，通过虚拟函数支持动态绑定，虚拟函数提供了一种封装类体系的实现细节的方法
面向对象的一些关键概念如下：
抽象(abstraction)：去除对象中不重要的细节，只有那些描述对象本质特征的关键点才被保留，是一种设计活动
类(class)：一种用户定义类型，必须允许程序员规定他所定义的类能够进行的操作，类里面的任何东西都是类的成员
对象(object)：某个类的一个特定变量，相当于类的实例
封装(encapsulation)：把类型、数据、函数组合在一起，组成一个类，例如 C 中的头文件就是一个非常脆弱的封装，它的组合形式是纯词法意义上的
继承(inheritance)：允许类从一个更简单的基类中接收数据结构和函数，派生类获得基类的数据和操作，并可以对它们进行改写，也可以在派生类中增加新的成员
C++中，类可以看做一个结构体，并且也可以用结构体来实现，类的通常形式是：
```
class  类名{
    访问控制：
```

声明

访问控制：

声明

};

访问控制是一个关键字，说明了谁可以访问接下来声明的数据或函数，有以下 2 种常见的：

public　属于 public 的声明在类的外部可见，并可按需要进行调用、操控，一般的原则是不要把类的数据做成 public，这样才符合面向对象编程的理论：只有类本身才能改变自己的数据，外部只能调用类的成员函数，这样保证了类的数据只会以合乎规则的方式被更新

private　属于 private 的声明只能被该类的成员函数使用，声明在类的外部可见(名字是已知的)，但是不能访问(比如说不能在类的外面直接打印它)

对象可以这样声明：　　类名 对象 1,对象 2,对象 3;

成员函数也叫方法，当它在类的外部声明时是这样的：

返回值　类名 ::函数名(参数列表){/*　实现　*/}

在类的内部实现时，要去掉类名和作用域运算符，如果::前面没有标识符，就表示查找范围是全局范围

调用方法使用圆点运算符：　对象 3.方法名();

每个成员函数都有一个隐藏的 this 指针参数，它允许对象在成员函数内部引用对象本身

当类的一个对象被声明时，构造函数被隐式调用，它负责对象的初始化。和它对应的有一个析构函数，当对象被销毁时它负责清理垃圾。

构造函数一般放在类里面，如果放在外面，写法是：

类名::类名(参数列表){/*　初始化操作　*/}

对应的析构函数则是在函数名前面加上一个符号‘~’

继承的写法是这样的：

class　派生类名 : 访问控制 基类名{ /*　声明　*/　};

如果我们要让对象支持一种自定义的运算，我们就要**重载**一个运算符，重载运算符在类的外部的写法是：

返回值 类名::operator 运算符(类名 &对象名){　/*自定义操作*/　}

隐藏的 this 指针允许我们引用运算符的左操作数，这里的右操作数是 f，它是类的一个实例，前面的&表示它是通过传递地址来调用的。

多态是指一个函数或运算符只有一个名字，但可以用于几个不同的派生类的能力。每个对象都实现该操作的一种变形，表现一种适合自己的行为。

当你想用派生类的成员函数取代基类的同名函数时，C++要求你预先通知编译器，通知的方法是在可能会被取代(形成多态)的基类的成员函数声明前面加上 virtual 关键字

7.水果类.Cpp

```
#include<iostream>
#include<cstdio>
using namespace std;
class Fruit{                                //基类
    public:
        virtual void peel();                //支持多态的成员函数
        void slice();
        juice();
        int operator+(Fruit &f);                //重载加法运算符
        Fruit(int i,int j){                     //构造函数
            weight=i;
            calories_per_oz=j;
         };
```

```
        virtual ~Fruit()=default;              //对析构函数进行动态绑定
    private:
        int weight,calories_per_oz;
};
void Fruit::peel(){
    cout<<"peeling a base class fruit!"<<endl;
}
void Fruit::slice(){
    printf("this ptr=%p\n",this);  //this 指针就指向那个将要调用该方法的对象
    this->weight--;                //只能在成员函数中修改 private 声明的变量
    this->calories_per_oz++;
    printf("right                                          now,banana's
weight=%d\ncalory=%d\n",this->weight,this->calories_per_oz);
}
int Fruit::operator+(Fruit &f){
    cout<<"calling fruit addition"<<endl;
    return weight+f.weight;
}
class Apple : public Fruit{            //派生类
    public:
        Apple():Fruit(0,0){            //继承基类的构造函数和参数
         };
        void make_candy_apple(float amount){       //派生类的新增成员函数
            printf("make candy with apple\n");
            cout<<amount*2<<endl;
        };
        void peel(){                       //和基类成员函数同名，但内容不同
            printf("peeling an apple!\n");
        };
};
int main(){
    Fruit melon(5,5),banana(10,10);        //对象声明时会隐式调用构造函数
    Apple green,red;
    Fruit *p;                     //指向对象的指针
    p=&green;
    p->peel();                    //派生类的方法取代了基类的同名方法
    p=&melon;
    p->peel();
    red.make_candy_apple(5.0);
    red.peel();
    banana.slice();
    int ounces=melon+banana;       //把两个对象的 weight 值相加
    cout<<ounces<<endl;
    return 0;
}
```

程序的运行结果如下：

```
peeling an apple!
peeling a base class fruit!
make candy with apple
10
peeling an apple!
this ptr=00000000006ffdf0
right now,banana's weight=9
```

```
calory=11
calling fruit addition
14
```

60.把 n 个球放进 m 个盒子，输出所有放法.C

```c
#include<stdio.h>
void f(int *r,int *p,int n,int m)
{
    if(m==1){
        for( ;r!=p;++r)
            printf("%d ",*r);
        printf("%d\n",n);
    }
    else
        for(*p=0;*p<=n;++*p)
            f(r,p+1,n-*p,m-1);
}
void g(int n,int m){
    int r[m];                //C99 的 VLA(Variable-Length_Arrays,变长数组)
    f(r,r,n,m);
}
int main(){
    g(3,5);
}
```

r 用于记录当前的结果，p 是写入该缓存的位置，那么 f(r,p,n,m)的意义就是把 n 个球放进 m 个盒子,如果只有一个盒子了就打印结果(最后一个盒子有 n 个球);如果有>1 个盒子，就放入 0 至 n 个球至当前的盒子，把余下的球和盒子以递归调用 f()

蒙特卡罗算法(Monte Carlo method)也称统计模拟方法，是一种以概率统计理论为指导的数值计算方法,法国数学家布丰提出的通过大量随机投针来求圆周率的实验被认为是蒙特卡罗算法的起源。我们在实现时就是用大量的随机数去轰炸一块区域，把落在某个区域内的点的数目作为该区域的面积或体积(实际上就是类似于高中学过的几何概型)

4.蒙特卡罗法求圆周率.java

```java
import java.util.Scanner;
public class Hello{
     static double MontePI(int n){
       double PI,x,y;
       int i,sum=0;
       for(i=1;i<n;i++){
           x=Math.random();
           y=Math.random();
           if((x*x+y*y)<=1){
               sum++;
           }
       }
       PI=4.0*sum/n;
       return PI;
     }
     public static void main(String[] args){
       int n;
       double PI;
       Scanner input=new Scanner(System.in);
       System.out.println("输入点的数量：");
       n=input.nextInt();
       PI=MontePI(n);
```

```
        System.out.println("PI="+PI);
      }
}
```

运行结果如下：

输入点的数量：

6666666

PI=3.1406349140634915

伪随机数是看似随机实则固定的周期性序列，生成随机数有硬件方法和软件方法，Linux 和 Windows 系统有自带的真随机数函数，它们是通过现实中的外部噪音实现的；通过确定的随机化算法生成的都是伪随机数，每一个种子数对应一个固定的伪随机值，是可以暴力破解的。这里有一个暴力破解出来的通过伪随机数输出 hello world 的程序：

5.Helloworld 程序(版本 2).java

```
import java.util.Random;
public class Helloworld {
      public static void main(String args[]){
        System.out.println(randomString(-229985452)+'
'+randomString(-147909649));
      }
      public static String randomString(int seed){
        Random rand=new Random(seed);
        StringBuilder sb=new StringBuilder();
        while(true){
          int n=rand.nextInt(27);
          if(n==0)
              break;
          sb.append((char)('`'+n));
        }
        return sb.toString();
      }
}
```

串的模式匹配是求模式串(子串)在文本串(主串)中的位置。暴力的解法非常简单，只要枚举文本串的起始位置 i，然后从该位开始逐位与模式串进行匹配，如果匹配过程中每一位都相同，则匹配成功；否则，只要出现某位不同，就把文本串的起始位置变为 i+1，并从头开始模式串的匹配。

```
//简单的模式匹配算法：返回串 t 在 s 中的位置，如果不匹配则返回-1
int strindex(char s[],char t[])
{
    int i,j,k;
    for(i=0;s[i]!='\0';i++)
    {
        for(j=i,k=0;t[k]!='\0'&&s[j]==t[k];j++,k++);
        if(k>0&&t[k]=='\0')
            return i;
    }
    return -1;
}
```

KMP 算法是改进的模式匹配算法，它借助于模式串的 next 数组，而 next 数组借助 PMT 数组来定义。PMT 中的值是字符串的前缀集合和后缀集合的交集中最长元素的长度，例如字符串 "ababa" 的前缀集合是 {"a","ab","aba","abab"}，后缀集合是 {"baba","aba","ba","a"}，交集为{"aba","a"}，其中最长元素是"aba"，长度为 3，所以字符串"ababa"的 PMT 数值是 3，将字符串[0 到 i,i++]的子串的 PMT 数值构成的 PMT

数组右移一位，第 1 位设为-1 就是 next 数组。

求 next 数组的过程就是模式串进行自我匹配的过程

KMP 算法的做法是:用 i 指示文本串,用 j 指示模式串，当发生失配(s[i]!=t[j])时，i 保持不变，j 退回到 next[j]并重新比较。

char:	a	b	a	b	a	b	c	a
index:	0	1	2	3	4	5	6	7
pmt:	0	0	1	2	3	4	0	1
next:	-1	0	0	1	2	3	4	0

假设主串和子串的长度分别为 m 和 n，KMP 算法将模式匹配的性能从 O(m×n)提升到了 O(m+n)，因为 KMP 算法减少了不必要的回溯。

61.KMP 模式匹配算法.C

```
#include<stdio.h>
#include<string.h>
int next[100];
void getnext(char *p,int *next){
    next[0]=-1;
    int i=0,j=-1,lenp=strlen(p);
    while(i<lenp){
        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,j=0,lent=strlen(t),lenp=strlen(p);
    while(i<lent&&j<lenp){
        if(j==-1||t[i]==p[j]){
            i++;j++;
        }
        else{
            if(j==0)
                i++;
            else
                j=next[j];
        }
    }
    return j==lenp?i-j:-1;
}
int main(void){
    char text[]="ababcdabcaabababcaabab";
    char pattern[]="abababca";
    getnext(pattern,next);
    for(int k=0;pattern[k]!='\0';k++)
        printf("%2d",next[k]);      //-1 0 0 1 2 3 4 0
    printf("%4d",KMP(text,pattern));            //10
}
```

快速幂是使用位运算的求幂算法，它将时间复杂度从 O(N)降到了 O(logN)，原理是把指数按二进制形式分解，比如$a^{11}=a^{2^0}\times a^{2^1}\times a^{2^3}$

```
double quickpow(double a,int n)
{
    if(n<0)
        return 1/quickpow(a,-n);
    double res=1.0;
```

```
    while(n)
    {
        if(n&1)
            res*=a;
        a*=a;
        n>>=1;
    }
    return res;
}
```

位运算还有一些应用，例如求一个整数的二进制表示中 1 的个数：

把一个整数减去 1，再和原整数做与运算，会把该整数最右边的一个 1 变成 0，那么一个整数的二进制表示中有多少个 1 就可以进行多少次这样的操作

```
int bitcount(int n)
{
    int count=0;
    while(n)
    {
        ++count;
        n=(n-1)&n;
    }
    return count;
}
```

补充 Python 的 4 个常用方法

```
>>> 'a b c d   '.rstrip()
'a b c d'
>>> 'a.b.c.d...'.rstrip('.')
'a.b.c.d'
>>> divmod(17,5)
(3, 2)
>>> bin(1000)
'0b1111101000'
>>> a = 2000
>>> a.bit_length()
11
```

下面介绍几个 windows 系统的操作：

在记事本前面一行加上“.LOG”可以显示之后每次自己修改或编辑的时间

Win+r 然后输入 psr.exe 可以打开步骤记录器，输入 snippingtool 可以使用截图工具，输入 cleanmgr 可以清理磁盘(删除系统认为没用的垃圾文件，不会误删有用文件)，输入 calc 可以临时使用计算器，输入 systeminfo 看电脑的配置信息

Ctrl+shift+esc 打开任务管理器，死机时可以找到你卡死的程序，然后点击结束进程即可

Ctrl+Z 可以撤回到上一步操作，ctrl+shift+T 可以恢复被误叉掉的网页

cmd 命令行还有更多玩法，假如你要隐藏 d 盘中的一个文件夹“我爱学习”，输入

attrib +s +a +h +r d:\我爱学习

并按下回车键即可，如果你想再把它恢复，则输入

attrib -s -a -h -r d:\我爱学习　并按下回车键即可。

放大镜：打开 win+加号　缩小 win+减号　关闭 win+esc

截取窗口：PrtSc 直接截取全屏，Alt+PrtSc 截取当前窗口，然后在 word 文档或者 qq 消息中可以直接粘贴此截图，如果想保存成文件，打开画图工具，再粘贴保存即可

C++编写的 atoi 函数：将字符串转为整型数

```
    int myAtoi(string str) {
        int i=0,base=0,sign=1;
        while(str[i]==' '){i++;}
        if(str[i]=='+'||str[i]=='-')
            sign=(1-2*(str[i++]=='-'));
        while(str[i]>='0' && str[i]<='9'){
            if(base>INT_MAX/10 || (base==INT_MAX/10 && (str[i]-'0')>7)){
                if(sign==1) return INT_MAX;
                else return INT_MIN;
            }
            base=10*base+(str[i++]-'0');
        }
        return sign*base;
    }
```

为了进一步说明某些位运算符，我们来看函数 getbits，它返回十进制整数 x 的二进制表示中从右边第 p 位(假定最右边是第 0 位)开始为头部向右截取的长度为 n 位的二进制字段所对应的十进制整数。

```
unsigned getbits(unsigned x,int p,int n)
{
    return (x>>(p+1-n)) & ~(~0<<n);
}
```

其中，表达式 x>>(p+1-n)将期望获得的字段移位到字的最右端，~0 的所有位都是 1，这里使用语句~0<<n 将~0 左移 n 位，并将最右边的 n 位用 0 填补，再使用~运算符对它按位取反，这样就建立了最右边 n 位全为 1 的屏蔽码。

树和森林可以用于实现一种简单的集合数据结构——**并查集**，它支持两种操作：合并和查找一种想法是可以使用树来表示集合，因为树上的每一个元素都有相同的根，这样就可以用根来命名这个集合。这些树不一定要是二叉树，开始时每个集合含有一个元素。我们需要的唯一信息就是树的根结点指针，因此可以设计一个数组 S，数组的每个成员 S[i]表示元素 i 的父亲，如果 i 是根，那么设定 S[i]=i

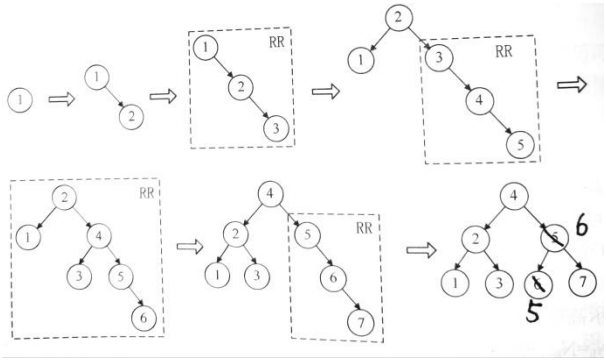
对集合中元素 X 的查找通过返回包含 X 的树的根而完成，具体实现方法就是反复寻找 X 的爹结点，直到找到根结点(即 S[X]=X 成立时)。合并两个元素所在的集合可以设计为：首先判断这两者是否属于同一集合(需要各自查找出它们的根结点，判断是否相同)，如果不属于，就把其中一棵树的根指针指向另一棵树的根结点。

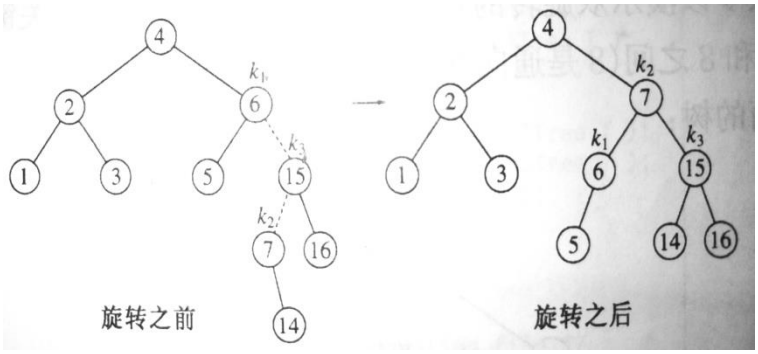
查找操作是可以通过路径压缩来优化的，把当前查询结点的路径上的所有结点的爹都指向根结点，就不需要一直回溯去找爹了，复杂度可以降到 O(1)

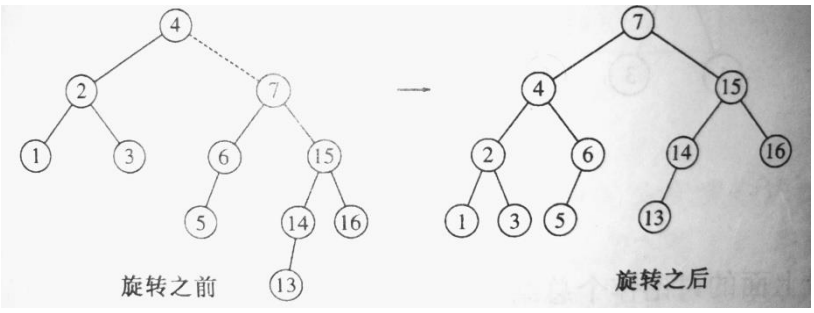
```
#define NumSets 20
int S[NumSets+1];
void Initialize()                //初始化数组 S
{
    int i;
    for(i=NumSets;i>0;i--)
        S[i]=i;
}
int Find(int X)                  //采用路径压缩的 Find 操作
{
    return X==S[X]?X:(S[X]=Find(S[X]));
}
void SetUnion(int a,int b)
{
    int Root1=Find(a),Root2=Find(b);
```

```
    if(Root1!=Root2)
        S[Root2]=Root1;
}
```

为了保持 AVL 树的平衡特性，面对新插入的结点，当插入发生在“外边”(左-左或右-右)时，需要执行单旋转；当插入发生在“内部”(左-右或右-左)时，需要执行稍微复杂一些的双旋转。这样改变了树的构造，但是 AVL 树本来就是用来提高查找效率的，只要保持平衡特性就能保持 O(logN)的平均查找时间，我们不用关心构造上的细节。







这是一张实用的表：

2 的幂	准确值	近似值	X 字节转换成 MB 等
7	128		
8	256		
10	1 024	一千	1K
16	65 536		64K
20	1 048 576	一百万	1MB
30	1 073 741 824	十亿	1GB
32	4 294 967 296		4GB
40	1 099 511 627 776	一万亿	1TB

定义一个空的类型，里面没有任何成员变量和成员函数，对它求 sizeof 会得到 1，是因为当我们声明该类型的实例时，它必须在内存中占有一定空间，否则将无法使用实例。如果在类型中增加一个构造函数和析构函数，对类型求 sizeof 的结果仍然是 1，因为调用这两个

函数只需知道函数的地址，而地址只和类型相关，与类型的实例无关。如果把析构函数标记为虚函数再对该类型求 sizeof，C++编译器一旦发现一个类型中有虚拟函数，就会为它生成虚函数表，并在该类型的每一个实例中添加一个指向虚函数表的指针，在 32 位的机器上，一个指针占 4 字节的空间，求 sizeof 得到 4；在 64 位的机器上，一个指针占 8 个字节的空间，求 sizeof 得到 8。C++可以用 struct 和 class 来定义类型，如果没有标明成员的访问权限级别，struct 中默认是 public，而 class 中默认是 private

面试问题：如何把字符串中的每个空格替换成”%20”？

最直接的做法是从头到尾扫描字符串，每遇到一个空格就把后面的串后移两格，并把空格替换掉，如果这样做，越是靠后的字符段被重复移动的次数越多，效率较低。

设计一个新的算法：因为替换后的字符串长度等于原来的长度加上空格数目乘以 2，所以我们可以准备两个指针 P1 和 P2，和原来的思路相反，把 P1 指向原来字符串的末尾，P2 指向替换之后的字符串的末尾。接下来向前移动 P1，逐个把 P1 指向的字符复制到 P2 指向的位置，直到碰到第一个空格为止。碰到空格之后，把 P1 向前移动一格，在 P2 前面插入字符串“%20”，同时把 P2 向前移动 3 格。重复这样的操作，直到 P1 和 P2 指向同一位置。

按照这样的思路，每个字符都只需移动一次，提高了效率。

面试问题：实现正则表达式匹配，支持'.' 和'*'。

'.' 匹配任何单个字符。

'*'匹配零个或多个前面的元素。

匹配应覆盖整个输入字符串（不是部分）。函数原型应该是：

bool isMatch（const char * s，const char * p）

一些例子：

isMatch（“aa”，“a”）→false

isMatch（“aa”，“aa”）→true

isMatch（“aaa”，“aa”）→false

isMatch（“aa”，“a *”）→true

isMatch（“aa”，“.*”）→true

isMatch（“ab”，“.*”）→true

isMatch（“aab”，“c * a * b”）→true

使用自底向上的动态规划解决：定义一个存储布尔值的 dp[i][j]，意思是 text[i:]和 pattern[j:]是否匹配？

```
    public boolean isMatch(String s, String p) {
        boolean[][] dp=new boolean[s.length()+1][p.length()+1];
        dp[s.length()][p.length()]=true;
        for(int i=s.length();i>=0;i--){
            for(int j=p.length()-1;j>=0;j--){
                boolean firstmatch=i<s.length() && (s.charAt(i)==p.charAt(j)
|| p.charAt(j)=='.');
                if(j<p.length()-1 && p.charAt(j+1)=='*'){
                    dp[i][j]=dp[i][j+2] || firstmatch && dp[i+1][j];
                }
                else{
                    dp[i][j]=firstmatch && dp[i+1][j+1];
                }
            }
        }
        return dp[0][0];
    }
```

反转二叉树(Java 实现)：

```
public class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;
    TreeNode(int x) { val = x; }
}
public TreeNode invertTree(TreeNode root) {
    if(root==null)
        return null;
    TreeNode tempright=root.right;
    root.right=invertTree(root.left);
    root.left=invertTree(tempright);
    return root;
}
```

下面给出链表相关操作的 3 个算法实现：(默认为非循环的单链表，且为 C 实现)

从尾到头打印链表：打印是一个只读操作，不能修改链表结构。我们分析一下，遍历链表只能从头开始，而输出的顺序则是相反的。也就是说第一个遍历到的结点最后一个输出，这符合“后进先出”的特点，自然联想到了栈，但是我们不想再编写一个栈实现，而递归正好就是栈结构，所以可以采用递归思路。每访问到一个结点的时候，先递归输出它的后继结点，再输出这个结点本身，这样就把输出结果反过来了。

在 O(1)时间内删除链表结点：常规的方法需要顺序查找，复杂度为 O(N)，如果设计 O(1)的算法呢？先把待删除结点 i 的后继结点 j 的数据复制到 i(同时覆盖了 i 的原有数据),然后把 i 的 next 指针指向 j 的后继结点，释放 j，这样做的效果恰好相当于删除了结点 i。为了让算法更加实用，我们还要考虑特殊情况，如果待删除的结点位于链表尾部，它没有后继结点，怎么办？这就只能按顺序从头开始遍历了。如果链表中只有一个结点，而我们需要删除这个结点，那就必须在释放结点之后，把链表的头结点指针设为 NULL。最后来分析一下总的复杂度为[(N-1)*O(1)+O(N)]/N，仍然是 O(1)

求链表中倒数第 k 个结点：简单的做法是把链表顺序遍历两次，第一次统计出链表中结点数目，需要走 n 步；第二次走 n-k+1 步找到倒数第 k 个结点。这样效率不高，我们来设计一个只用遍历一次的算法，和前面替换字符串空格的思路一样，用双指针法。指针 P1 先从链表的头指针开始向前走 k-1 步，指针 P2 保持不动；然后从指针 P1 的第 k 步开始，指针 P2 也开始从链表的头指针开始移动，两个指针保持同步移动，保持距离为 k-1；最后当 P1 到达链表的尾结点时，P2 恰好指向倒数第 k 个结点。为了提高代码的鲁棒性，我们要处理可能会引起程序崩溃的问题，例如输入的头指针为空，程序会试图访问空指针指向的内存，崩溃；例如链表的结点总数小于 k，指针向前走 k-1 步就会发生错误；例如输入的 k 为 0，而我们要求 k 是无符号整数，k-1 不是-1，而是 4294967295(无符号的 0xFFFFFFFF)，for 循环将要执行 4294967295 次，这是必须避免的问题。让函数返回 NULL 指针以及在循环体前面增加判断条件来即可。

```
#include<stdlib.h>
#include<stdio.h>
struct Node;
typedef struct Node *PtrToNode;
typedef PtrToNode Position;
struct Node{
    int data;
    Position next;
```

```
};
void PrintReversingly(Position Head)
{
    if(Head!=NULL)
    {
        if(Head->next!=NULL)
            PrintReversingly(Head->next);
        printf("%d\t",Head->data);
    }
}
void DeleteNode(Position Head,Position ToBeDeleted)
{
    if(!Head||!ToBeDeleted)
        return;
    if(ToBeDeleted->next!=NULL)
    {
        Position NextNode=ToBeDeleted->next;
        ToBeDeleted->data=NextNode->data;
        ToBeDeleted->next = NextNode->next;
        free(NextNode);
    }
    else if(Head==ToBeDeleted)
    {
        free(ToBeDeleted);
        Head=NULL;
    }
    else
    {
        Position One=Head;
        while(One->next!=ToBeDeleted)
            One=One->next;
        One->next=NULL;
        free(ToBeDeleted);
    }
}
Position FindKthToTail(Position Head,unsigned int k)
{
    if(Head==NULL||k==0)
        return NULL;
    Position Ahead=Head;
    Position Behind=NULL;
    for(unsigned int i=0;i<k-1;i++)
    {
        if(Ahead->next!=NULL)
            Ahead=Ahead->next;
        else
            return NULL;
    }
    Behind=Head;
    while(Ahead->next!=NULL)
    {
        Ahead=Ahead->next;
        Behind=Behind->next;
    }
    return Behind;
}
```

```
int main(){return 0;}
```

反转链表的两种方法（迭代和递归）：

```
struct ListNode {
    int val;
    struct ListNode *next;
};
struct ListNode* reverseList(struct ListNode* head) {
    if(NULL==head) return head;
    struct ListNode *p=head;
    p=head->next;
    head->next=NULL;
    while(NULL!=p){
        struct ListNode *ptmp=p->next;
        p->next=head;
        head=p;
        p=ptmp;
    }
    return head;
}
struct ListNode* reverseListRe(struct ListNode* head) {
    if(NULL==head||NULL==head->next) return head;
    struct ListNode *p=head->next;
    head->next=NULL;
    struct ListNode *newhead=reverseListRe(p);
    p->next=head;
    return newhead;
}
```

寻找两个链表的第一个公共结点：

```
struct ListNode {
    int val;
    struct ListNode *next;
};
struct  ListNode  *getIntersectionNode(struct  ListNode  *headA,  struct
ListNode *headB) {
    if( !headA || !headB)
        return NULL;
    struct ListNode *a = headA, *b = headB;
    while(a != b){
        a = a ? a->next : headB;
        b = b ? b->next : headA;
    }
    return a;
}
```

给定一个数组 nums，有一个大小为 k 的滑动窗口，它从数组的最左边移动到最右边。 您只能在窗口中看到 K 个数字。 每次滑动窗口向右移动一个位置

```
Window position                Max
---------------               -----
[1  3  -1] -3  5  3  6  7       3
 1 [3  -1  -3] 5  3  6  7       3
 1  3 [-1  -3  5] 3  6  7       5
 1  3  -1 [-3  5  3] 6  7       5
 1  3  -1  -3 [5  3  6] 7       6
 1  3  -1  -3  5 [3  6  7]      7
```

例如，给定 nums = [1,3，-1，-3,5,3,6,7]，k = 3。返回最大滑动窗口为[3,3,5,5,6,7]
使用 Java 中的双端队列，在每个步骤中，deque 的头部是[i-（k-1），i]中的最大元素，

```
    public int[] maxSlidingWindow(int[] nums, int k) {
        if(nums==null || k<=0)
            return new int[0];
        Deque<Integer> q=new ArrayDeque<Integer>();
        int n=nums.length;
        int[] res=new int[n-k+1];
        int ri=0;
        for(int i=0;i<n;i++){
            while(!q.isEmpty() && q.peek()<i-k+1)
                q.poll();
            while(!q.isEmpty() && nums[q.peekLast()]<nums[i])
                q.pollLast();
            q.offer(i);
            if(i-k+1>=0)
                res[ri++]=nums[q.peek()];
        }
        return res;
    }
```

由于图论问题渗透整个计算机科学，图算法在算法领域中至关重要，首先我们来看如何在计算机中表示一张图

图 G 由顶点集 V 和边集 E 组成，记作 G=(V,E)，图中的顶点数|V|又称为图 G 的阶，边集 E={(x,y)|x∈V,y∈V}，图的顶点集一定非空，边集可以为空。边可以分为有向和无向的，有向边也叫弧，记作<x,y>，x 是弧尾，y 是弧头，方向是 x→y，x 邻接到 y，y 邻接自 x 无向边记作(x,y)，x 和 y 的次序任意。根据组成图的边集的元素是弧还是边，可以把图分为有向图和无向图。简单图的特点是：两个顶点之间不存在重复边，且，一个顶点不存在连接自己到自己的边。简单图的反义词是多重图。无向完全图：任意两个顶点之间都有边。有向完全图：任意两个顶点之间都有方向相反的两条弧。阶为 n 的无向完全图有 n(n-1)/2 条边。要保证阶为 n 的无向图在任何情况下都是连通的，至少需要(n-1)(n-1-1)/2+1 条边。并非 V 和 E 的任何子集都能构成 G 的子图。

在无向图中，若两个顶点 x,y 之间有路径存在，则称 x 和 y 是连通的。若图 G 中任意两个顶点都是连通的，那么图 G 就是一个连通图，否则就是非连通图。无向图中的极大连通子图称为连通分量，无向图中包含全部顶点的极小连通子图称为**生成树**，阶为 n 的无向图的生成树有 n-1 条边，加一条边就会产生环，少一条边就会不再连通。这里的极大极小是指包含的边数(包含全部或最少)。图中边数的少或多可以分别用两个形容词稀疏和稠密来描述。路径上边的数目就是路径长度，第一个顶点和最后一个顶点相同的路径称为回路(环)，顶点不重复出现的路径叫做简单路径。

如果阶为 n 的图有大于 n-1 条边，那么图中一定存在环

在有向图中，若从顶点 x 到 y 以及从顶点 y 到 x 都有路径存在，就说 x 和 y 是强连通的。

无向图中依附于某顶点 k 的边的数目称为 k 的度，全部顶点的度的总和=总边数*2

有向图中顶点的度分为入度和出度，全部顶点的入度总和=全部顶点的出度总和=总边数

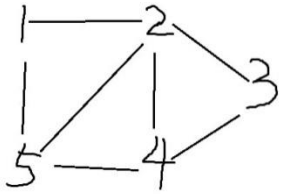
在每条边边上标一个数，就把图变成了带权图(网)

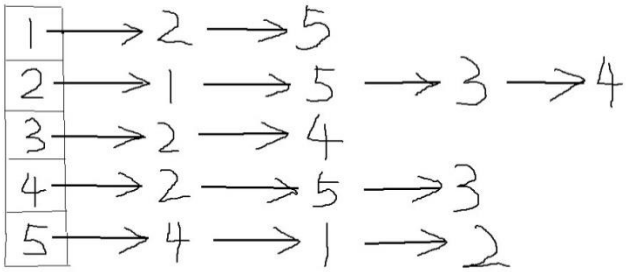
在计算机中表示一张图有两种主要方式：**邻接矩阵**(顺序存储)和**邻接表**(链式存储)

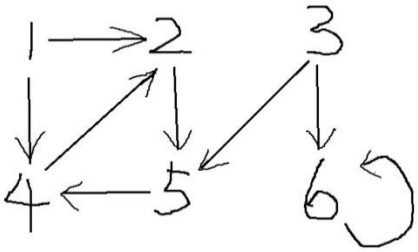
要表示图，首先要把图中的顶点编号，假设编号为 1,2,3,..,i,..,j,..n，阶为 n 的图的邻接矩阵 A 也是一个 n 阶的方阵，其中元素 A[i][j]=

0　　如果(i,j)或<i,j>不是边集中的元素

1　　如果(i,j)或<i,j>是边集中的元素







$$\begin{bmatrix} 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

如果是网，就把矩阵中的 1 改成边上的权值，0 改成∞

无向图和无向网的邻接矩阵一定是一个对称矩阵，并且对于无向图的邻接矩阵，第 i 行或第 i 列中的非 0 非∞元素的个数=顶点 i 的度

有向图的邻接矩阵中，第 i **行**或第 i **列**中的非 0 非∞元素的个数=顶点 i 的**出**度或**入**度

设图 G 的邻接矩阵是 A，A^n的元素$A^n[i][j]$等于由顶点 i 到顶点 j 的长度为 n 的路径的条数，证明方法参见离散数学教材

邻接表的构造方法是，把图中所有顶点放在一个结构数组(顶点表)中，对于其中每一个顶点，把这个顶点重新作为一个新的单链表(边表)的出发点(结构数组的元素中包含边表的头指针)，把和这个顶点相邻接的所有顶点放在这个顶点的边表中(放的次序任意，取决于建立邻接表的算法)

假设有 n 个顶点 e 条边的有向图采用邻接表存储，那么删除与某个顶点 v 相关的所有边的操作如下：先删除下标为 v 的顶点表结点的单链表，再扫描所有边表结点，删除所有顶点 v 的入边，总的复杂度为 O(n+e)

图的遍历要求访问图中所有顶点且每个顶点只访问一次，对于 n 个顶点 e 条边的图，采用不同的存储方式并进行深度优先搜索和广度优先搜索的算法性能比较如下：

	DFS 时间复杂度	DFS 空间复杂度	BFS 时间复杂度	BFS 空间复杂度
邻接表	O(n+e)	O(n)	O(n+e)	O(n)
邻接矩阵	O(n^2)		O(n^2)	

8.判断有向图中是否存在回路.Cpp

假设以邻接矩阵作为图的存储结构，编写算法判断在给定的有向图中是否存在一个简单的有向回路，若存在，则以顶点序列的方式输出该回路(找到一条即可)

可以通过深度优先搜索的方法实现，从编号为 0 的顶点出发，若两个顶点间存在路径，则记录下源顶点，并将该顶点标记为已访问(标记为-1)，在遍历的过程中，若有顶点与源顶点相同，则说明存在环

```
#include<iostream>
#include<string.h>
using namespace std;
const int N=100;
int G[N][N];            //0 means cycle doesn't exist,1 means cycle exist
```

```
int path[N],visited[N],n,cycle;
int DFS(int u,int start)        //depth-first-search a directed graph
{
    int i;
    visited[u]=-1;          //vertex u signed as already been traversed
    path[u]=start;          //record source vertex
    for(i=0;i<n;i++)
    {
        if(G[u][i]&&i!=start)
                    //there is a edge between u and i,and i is not start
        {
            if(visited[i]<0)
            {
                cycle=u;
                return 0;
            }
            if(!DFS(i,u))          //if path exist,searching continue
                return 0;
        }
    }
    visited[u]=1;           //if there is no edge between u and i
    return 1;               //stop searching
}
void DisPath(int u)         //print the vertex in cycle
{
    if(u<0)
        return;
    DisPath(path[u]);
    cout<<" "<<u;
}
int main(){
    int i,j;
    cout<<"please enter the number of vertexs in graph:"<<endl;
    cin>>n;
    memset(G,0,sizeof(G));
    cout<<"please enter a"<<n<<"*"<<n<<"matrix:"<<endl;
    for(i=0;i<n;i++)
        for(j=0;j<n;j++)
            cin>>G[i][j];
    cycle=-1;
    for(i=0;i<n;i++)
        if(!visited[i]&&!DFS(i,-1))     //vertex i is not been traversed yet
            break;
    if(cycle<0)
        cout<<"cycle doesn't exist!"<<endl;
    else
    {
        cout<<"cycle exist!"<<endl;
        DisPath(cycle);
        cout<<endl;
    }
}
```

运行结果如下：

9.图的广度优先遍历.Cpp

假设一个无向图按照邻接表形式存储，编写 BFS 的算法。

```
please enter the number of vertexs in graph:
4
please enter a4*4matrix:
0 1 1 0
0 0 1 1
0 0 0 0
1 0 0 0
cycle exist!
 0 1 3
```

定义一个数组 visited[]，用来标记顶点是否已经被访问，这个标记数组被初始化为 0，表示顶点未被访问；为 1 时表示顶点已经被访问。从第一个顶点 v0 出发，访问该顶点并设置 visited[v0] 为 1，然后将 v0 入队。若队列不为空，将队头元素(顶点)出队，依次访问该顶点的所有邻接点，同时将这些顶点标记为已经访问，并将其邻接顶点依次入队。重复以上操作，直到图中的所有顶点都已经被访问过。

```
#include<stdlib.h>
#include<iostream>
#include<string.h>
using namespace std;
//图的邻接表类型定义
typedef char VertexType[4];
typedef char InfoPtr;
typedef int VRType;
#define MAXSIZE 100                                  //最大顶点个数
typedef enum{DG,DN,UG,UN}GraphKind; //图的类型:有向图,有向网,无向图,无向网
typedef struct ArcNode                               //边结点的类型定义
{
    int adjvex;                                  //弧指向的顶点的位置
    InfoPtr *info;                                   //与弧相关的信息
    struct ArcNode *nextarc;          //指示下一个与该顶点相邻接的顶点
}ArcNode;
typedef struct VNode                   //头结点的类型定义
{
    VertexType data;                     //用于存储顶点
    ArcNode *firstarc;                //指示第一个与该顶点邻接的顶点
}VNode,AdjList[MAXSIZE];
typedef struct                                   //图的类型定义
{
    AdjList vertex;
    int vexnum,arcnum;                     //图的顶点数目和弧的数目
    GraphKind kind;                          //图的类型
}AdjGraph;
int LocateVertex(AdjGraph G,VertexType v)     //返回图中顶点对应的位置
{
    int i;
    for(i=0;i<G.vexnum;i++)
        if(strcmp(G.vertex[i].data,v)==0)
            return i;
    return -1;
}
void CreateGraph(AdjGraph *G)                 //采用邻接表创建无向图
{
    int i,j,k;
    VertexType v1,v2;                            //定义两个顶点
    ArcNode *p;
    cout<<"请输入图的顶点数和边数：";
    cin>>(*G).vexnum>>(*G).arcnum;
```

```
    cout<<"请输入"<<G->vexnum<<"个顶点的值："<<endl;
    for(i=0;i<G->vexnum;i++)                    //将顶点存储在头结点中
    {
        cin>>G->vertex[i].data;
        G->vertex[i].firstarc=NULL;             //将相关联的顶点置为空
    }
    cout<<"请输入弧尾 弧头："<<endl;
    for(k=0;k<G->arcnum;k++)                        //建立边链表
    {
        cin>>v1>>v2;
        i=LocateVertex(*G,v1);                    //确定 v1 对应的编号
        j=LocateVertex(*G,v2);                   //确定 v2 对应的编号
                                  //j 为弧头 i 为弧尾创建邻接表
        p=(ArcNode*)malloc(sizeof(ArcNode));
        p->adjvex=j;
        p->info=NULL;
        p->nextarc=G->vertex[i].firstarc;
        G->vertex[i].firstarc=p;
                                  //i 为弧头 j 为弧尾创建邻接表
        p=(ArcNode*)malloc(sizeof(ArcNode));
        p->adjvex=i;
        p->info=NULL;
        p->nextarc=G->vertex[j].firstarc;
        G->vertex[j].firstarc=p;
    }
    (*G).kind=UG;
}
void DestroyGraph(AdjGraph *G)                      //销毁无向图
{
    ArcNode *p,*q;
    for(int i=0;i<(*G).vexnum;i++)           //释放图中的边表结点
    {
        p=G->vertex[i].firstarc;          //p 指向边表的第一个结点
        if(p!=NULL)             //如果边表不为空，则释放边表的结点
        {
            q=p->nextarc;
            free(p);
            p=q;
        }
    }
    (*G).vexnum=0;                           //将顶点数设为 0
    (*G).arcnum=0;                             //将边数设为 0
}
void BFS(AdjGraph G)                //广度优先非递归遍历无向图
{
    int v,front,rear,visited[MAXSIZE];
    int queue[MAXSIZE];                           //定义队列
    ArcNode *p;
    front=rear=-1;                               //初始化队列
    for(v=0;v<G.vexnum;v++)                       //初始化标志位
        visited[v]=0;
    v=0;
    visited[v]=1;                 //设置访问标志为 1，表示已经被访问过
    cout<<G.vertex[v].data<<" ";
```

```
    rear=(rear+1)%MAXSIZE;
    queue[rear]=v;                                  //v 入队
    while(front<rear)                               //如果队列不空
    {
        front=(front+1)%MAXSIZE;
        v=queue[front];                         //队头元素出队赋给 v
        p=G.vertex[v].firstarc;
        while(p!=NULL)                      //遍历序号为 v 的所有邻接点
        {
            if(visited[p->adjvex]==0)        //如果该顶点没有被访问过
            {
                visited[p->adjvex]=1;
                cout<<G.vertex[p->adjvex].data<<" ";
                rear=(rear+1)%MAXSIZE;
                queue[rear]=p->adjvex;
            }
            p=p->nextarc;                   //指向下一个邻接点
        }
    }
}
void DisplayGraph(AdjGraph G)                       //输出图的邻接表
{
    int i;
    ArcNode *p;
    cout<<G.vexnum<<"个顶点："<<endl;
    for(i=0;i<G.vexnum;i++)
        cout<<G.vertex[i].data<<" ";
    cout<<endl<<2*G.arcnum<<"条边："<<endl;
    for(i=0;i<G.vexnum;i++)
    {
        p=G.vertex[i].firstarc;
        while(p)
        {
            cout<<G.vertex[i].data<<"→"<<G.vertex[p->adjvex].data<<" ";
            p=p->nextarc;
        }
        cout<<endl;
    }
}
int main(){
    AdjGraph G;
    cout<<"采用邻接表创建无向图 G："<<endl;
    CreateGraph(&G);
    cout<<"输出无向图 G 的邻接表："<<endl;
    DisplayGraph(G);
    cout<<"广度优先遍历无向图 G："<<endl;
    BFS(G);
    cout<<endl;
    DestroyGraph(&G);}
```

运行结果如下：

```
采用邻接表创建无向图G:
请输入图的顶点数和边数: 10 10
请输入10个顶点的值:
a b c d e f g h i j
请输入弧尾 弧头:
a b
a d
a c
b e
e h
e i
i f
f j
f d
d g
输出无向图G的邻接表:
10个顶点:
a b c d e f g h i j
20条边:
a→c a→d a→b
b→e b→a
c→a
d→g d→f d→a
e→i e→h e→b
f→d f→j f→i
g→d
h→e
i→f i→e
j→f
广度优先遍历无向图G:
a c d b g f e j i h
```

给定图 G=(V,E)和一个可以识别的源结点 s，广度优先搜索能够计算从 s 到每个可到达的结点的距离(最少的边数)，同时生成一棵广度优先搜索树，该树以 s 为根结点，包含所有可以从 s 到达的结点 v，在这棵树里从结点 s 到结点 v 的简单路径所对应的就是图 G 中从结点 s 到结点 v 的最短路径。该算法既适用于无向图也适用于有向图。

对于一个有向无环图 G，其**拓扑排序**是 G 中所有结点的一种线性次序，该次序满足的条件的是：如果图中包含弧<u,v>，则 u 在 v 的前面(如果 G 中存在环，则不可能排出一个线性次序)。

为了理解“访问结点”这个过程，可以把结点加上三种颜色白灰黑来标记时间戳，每个结点的初始颜色都是白色，在结点被发现后变为灰色，在其邻接链表被扫描完成后变为黑色。时间戳 $v.d$ 记录 v 被涂上灰色的时刻，$v.f$ 记录 v 被涂上黑色的时刻。

下面的简单算法可以对一个有向无环图进行拓扑排序：

```
TOPOLOGICAL-SORT(G)
call DFS(G) to compute finishing times v.f for each vertex v
as each vertex is finished,insert it onto the front of a linked list
return the linked list of vertices
```

对于连通无向网 G(V,E)和权重函数 ω:E→**R**，假设我们希望找到 G 的一个无环子图 T，使得 T 既能把 G 中所有结点连接起来，又使得 T 中所有边的权重之和最小，那么子图 T 称为 G 的**最小生成树**，求解这个问题有一个通用的策略：

在每一步生长最小生成树的一条边，并在整个策略的实施过程中，管理一个遵守这样的循环不变式的边的集合 A：<u>在每一遍循环之前，A 是某棵最小生成树的一个子集</u>。在每一步，我们要做的就是选择一条边(u,v)，把它加入到集合 A 中，使得 A 不违反循环不变式，即 A∪{(u,v)}也是某棵最小生成树的子集。我们把这样的边称为边集合 A 的安全边。

求解最小生成树问题有两个经典算法:**Kruskal 算法**和 **Prim 算法**,它们都是基于贪心策略，两者的区别仅在于找到安全边的方法不同

在 Kruskal 算法中，集合 A 是一个森林，首先将 A 初始化为一个空集合，并创建|V|棵树，每棵树只包含一个结点。然后使用一个循环按照权重从低到高的次序对每一条边逐一检查，对于每一条边(u,v)来说，该循环检查端点 u 和 v 是否属于同一棵树，如果是，这条边不能加入到森林中(否则将形成环)；如果不是，则把它加入集合 A 中，并将两棵树中的结点合并。可见，该算法还要依赖于并查集数据结构的实现

在 Prim 算法中，集合 A 是一棵树，这棵树从任意的一个根结点开始，一直生长到覆盖 V 中所有顶点为止，每一步在连接集合 A 和 A 之外的结点的所有边中，选择一条轻量级边(横跨两个点集的所有边中，权重最小的边)，执行时，可以把所有不在树 A 中的结点都存放在一个最小优先队列中。

还有一种常用的求解最小生成树的方法——破圈法：任取一个环，去掉环上权重最大的边，

反复执行这一步骤，见圈就破，直到图中没有圈为止。

最短路径是图论中一个很经典的问题：给定图 G(V,E)，求一条从起点到终点的路径，使得这条路径上经过的所有边的权重的和最小。可见，最短路径一定是简单路径。解决此问题的经典算法有 **Dijkstra 算法**，**Bellman-Ford 算法**，**Floyd-Warshall 算法**

Dijkstra 算法用来解决单源最短路问题，即给定起点 s，求从 s 出发到达其他每个顶点的最短距离。该算法唯一不适用的情况是权重为负。基本思想是：先把图中所有边抹去，只有当从起点出发并占领某个顶点后才把从这个顶点发出的边显现出来，并且在已经占领的顶点上标记从起点到这个顶点所需要的最短距离。首先我们占领了起点，每一次都从还未占领的顶点中选择当前距离起点最近的那一个(设为 k)，然后显现从 k 出发的所有边，并查看以 k 为中间点的情况下，能否使从起点到达另外某些还未占领的顶点的最短距离变小，如果能，就把那个还未占领的顶点最短距离的标记用这个较小的值更新(称为“松弛”操作)。不断重复，直到占领图中全部顶点。

Bellman-Ford 算法也可以解决单源最短路问题，并且适用于边的权重为负的情况，基本思想是：设置一个数组 d[]，用来存放从起点到达各个到达的最短距离，同时返回一个 bool 值：如果图中存在可以从起点到达的负环(环中边的权重之和为负)，那么返回 false(因为这样会把最短路径变短，影响求解)，否则返回 true。对图中的边执行|V|-1 轮操作，每一轮都遍历图中所有边：对每一条边 u→v，如果以 u 为中间点可以使 d[v]变小，就执行松弛操作。可见该算法的时间复杂度为 O(|V||E|)。

Floyd 算法用于解决全源最短路问题，即求解图中任意两个顶点之间的最短路径长度，基本思想是：枚举顶点 k，以顶点 k 作为中间点，枚举所有顶点对 i,j，如果 k 能使得从 i 到 j 的路径长度缩短，就执行松弛操作。

如果用有向无环图(DAG 图)表示一个工程，其顶点表示活动，用有向边表示活动的先后顺序，这种顶点表示活动的网络称为 AOV 网。

如果在有向网中，以顶点表示事件(比如活动完成之后的奖励大会)，有向边表示活动，边上的权重表示完成活动的开销，这种用边表示活动的网络称为 AOE 网。它有且只有一个入度为 0 的顶点(工程的开始，源点)和一个出度为 0 的顶点(工程的结束，汇点)，从开始到结束的所有路径中最长的一条称为关键路径，关键路径上的活动称为关键活动。完成整个工程的最短时间就是关键路径的长度。关键路径可能有多条，只有所有关键路径上的活动时间同时减少时，才能缩短工期。求关键路径的方法是以拓扑排序为基础的。

对一个 DAG 图进行拓扑排序的通用方法是：从中选择入度为 0 的顶点，它们的次序任意，可以先暂存在栈或队列中，然后在图中删除该顶点和所有从该顶点出发的有向边，不断重复前两步操作直到图中剩下出度为 0 的顶点为止。

如果一个有向图具有有序的拓扑排序序列，那么它的邻接矩阵必定为三角形矩阵

10.图的最短路径问题.Cpp

```
#include<cstdio>
#include<algorithm>
using namespace std;
const int MAXV=200;                          //最大顶点数
const int INF=1000000000;                    //设 INF 表示无穷大
int n,m,s,G[MAXV][MAXV];                     //n:顶点数,m:边数,s:起点
int d[MAXV];                                 //起点到达各点的最短路径长度
int dis[MAXV][MAXV];                         //dis[i][j]表示顶点 i 和 j 的最短距离
bool vis[MAXV]={false};                      //标记数组，vis[i]==true 表示已访问
void Dijkstra(int s)
{
    fill(d,d+MAXV,INF);                      //将整个 d 数组赋值为 INF，慎用 memset
```

```
    d[s]=0;                             //起点到达自身的距离为0
    for(int i=0;i<n;i++)
    {
        int u=-1,MIN=INF;       //u使d[u]最小，MIN存放该最小的d[u]
        for(int j=0;j<n;j++)        //找到未访问的顶点中d[]最小的
        {
            if(vis[j]==false&&d[j]<MIN)
            {
                u=j;
                MIN=d[j];
            }
        }
        if(u==-1)                   //找不到小于INF的d[u]
            return;                //说明剩下的顶点和起点s不连通
        vis[u]=true;                        //标记u为已访问
        for(int v=0;v<n;v++)
                //如果v未访问且u能到达v且以u为中间点可以使d[v]更小
            if(vis[v]==false&&G[u][v]!=INF&&d[u]+G[u][v]<d[v])
                d[v]=d[u]+G[u][v];               //执行松弛操作
    }
}
void Floyd()
{
    for(int k=0;k<n;k++)                          //枚举顶点
        for(int i=0;i<n;i++)
            for(int j=0;j<n;j++)

if(dis[i][k]!=INF&&dis[k][j]!=INF&&dis[i][k]+dis[k][j]<dis[i][j])
                  dis[i][j]=dis[i][k]+dis[k][j];  //松弛操作
}
int main()
{
    int u,v,w;
    scanf("%d%d%d",&n,&m,&s);          //输入顶点个数，边数，起点的编号
    fill(G[0],G[0]+MAXV*MAXV,INF);                      //初始化图G
    fill(dis[0],dis[0]+MAXV*MAXV,INF);              //初始化dis数组
    for(int i=0;i<n;i++)
        dis[i][i]=0;                   //顶点i到顶点i的距离初始化为0
    for(int i=0;i<m;i++)
    {
        scanf("%d%d%d",&u,&v,&w);        //输入顶点u,v以及边u→v的权重
        G[u][v]=w;
        dis[u][v]=w;                        //以有向图为例进行输入
    }
    Dijkstra(s);
    printf("Dijkstra:\n");
    for(int i=0;i<n;i++)
        printf("%d ",d[i]);
    printf("\n");
    printf("Floyd:\n");
    Floyd();
    for(int i=0;i<n;i++)
    {
        for(int j=0;j<n;j++)
            printf("%d ",dis[i][j]);
```

```
        printf("\n");
    }
    return 0;
}
```

程序的运行结果如下：

```
6 8 0
0 1 1
0 3 4
0 4 4
1 3 2
2 5 1
3 2 2
3 4 3
4 5 3
Dijkstra:
0 1 5 3 4 6
Floyd:
0 1 5 3 4 6
1000000000 0 4 2 5 5
1000000000 1000000000 0 1000000000 1000000000 1
1000000000 1000000000 2 0 3 3
1000000000 1000000000 1000000000 1000000000 0 3
1000000000 1000000000 1000000000 1000000000 1000000000 0
```

现在可以继续学 Python 了，为了发挥它的强大功能，你需要在电脑上安装这两个安装包

你可以在互联网上下载它们，是完全免费的

pycharm-community-2017.2.3　　这是 python 的一个 IDE

Anaconda2-4.4.0-Windows-x86_64　　这是一个包含大量实用的第三方库(轮子)的包

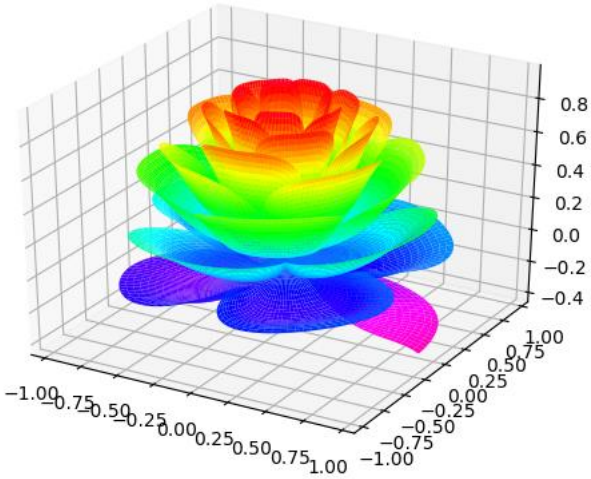
如果你不会使用某个库(例如 numpy)，你可以在 Python 命令行输入 help(numpy)

在命令行输入 import this，你可以领略 Python 的哲学：

```
import this
The Zen of Python, by Tim Peters
Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

7.画一朵花.py

```
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
from matplotlib.ticker import LinearLocator
import matplotlib.pyplot as plt
import numpy as np
fig = plt.figure()
ax = fig.gca(projection='3d')
[x,  t]  =  np.meshgrid(np.array(range(25))/24.0,  np.arange(0,  575.5,
0.5)/575*17*np.pi-2*np.pi)
p = (np.pi/2)*np.exp(-t/(8*np.pi))
u = 1-(1-np.mod(3.6*t, 2*np.pi)/np.pi)**4/2
y = 2*(x**2-x)**2*np.sin(p)
r = u*(x*np.sin(p)+y*np.cos(p))
surf         =         ax.plot_surface(r*np.cos(t),         r*np.sin(t),
u*(x*np.cos(p)-y*np.sin(p)), rstride=1, cstride=1, cmap=cm.gist_rainbow_r,
linewidth=0, antialiased=True)
plt.show()                          #
```




```
        THON!PYTH          YTHON!PYT
    !PYTHON!PYTHON!PY   N!PYTHON!PYTHON!P
  N!PYTHON!PYTHON!PYTHON!PYTHON!PYTHON!PYTH
 N!PYTHON!PYTHON!PYTHON!PYTHON!PYTHON!PYTHON
N!PYTHON!PYTHON!PYTHON!PYTHON!PYTHON!PYTHON!P
!PYTHON!PYTHON!PYTHON!PYTHON!PYTHON!PYTHON!PY
PYTHON!PYTHON!PYTHON!PYTHON!PYTHON!PYTHON!PYT
YTHON!PYTHON!PYTHON!PYTHON!PYTHON!PYTHON!PYTH
THON!PYTHON!PYTHON!PYTHON!PYTHON!PYTHON!PYTHO
HON!PYTHON!PYTHON!PYTHON!PYTHON!PYTHON!PYTHON
 N!PYTHON!PYTHON!PYTHON!PYTHON!PYTHON!PYTHON
  PYTHON!PYTHON!PYTHON!PYTHON!PYTHON!PYTHON
  YTHON!PYTHON!PYTHON!PYTHON!PYTHON!PYTHON!
    ON!PYTHON!PYTHON!PYTHON!PYTHON!PYTHON
     !PYTHON!PYTHON!PYTHON!PYTHON!PYTHON
      YTHON!PYTHON!PYTHON!PYTHON!PYTHON
        ON!PYTHON!PYTHON!PYTHON!PYTHO
          PYTHON!PYTHON!PYTHON!PYTH
            HON!PYTHON!PYTHON!PYT
              PYTHON!PYTHON!P
                ON!PYTHON
                  YTH
                   H
```

8.画一个字符拼成的心形.py

```
print
'\n'.join(['''.join([('PYTHON!'[(x-y)%7]if((x*0.05)**2+(y*0.1)**2-1)**3-
(x*0.05)**2*(y*0.1)**3<=0  else'  ')for  x  in  range(-30,30)])for  y  in
range(15,-15,-1)])
```

9.list 的分组和碾平.py

```
a = [3, 6, 5, 8, 4, 0, 9, 7, 8, 1, 3, 2, 6, 3]
b = [a[i:i+3] for i in xrange(0, len(a), 3)]
print b
flatten = lambda x: [y for l in x for y in flatten(l)] if type(x) is list
else [x]
c = flatten(b)
print c
```

运行结果如下：

```
[[3, 6, 5], [8, 4, 0], [9, 7, 8], [1, 3, 2], [6, 3]]
[3, 6, 5, 8, 4, 0, 9, 7, 8, 1, 3, 2, 6, 3]
```

可以用这样的方法直接计算表达式的值：

```
>>> print input()
2+3*9**2-8*7
189
```

求 2 的 1000 次方的各位数之和：

```
print sum(map(int, str(2**1000)))
```

一行筛质数：

```
print ('\n'.join([str(item) for item in filter(lambda x: all(map(lambda p: x % p != 0, range(2, x))), range(2, 1001))]))
```

join 函数的用法是　'sep'.join(seq)

其中 sep 是分隔符，seq 是要连接的元素序列、字符串、元组、字典

上述语句的功能是：以 sep 作为分隔符，把 seq 的所有元素合并成一个新的字符串并返回

map()函数接收两个参数，第 1 个参数是函数，第 2 个参数是序列，map()把传入的函数依次作用到序列的每一个元素，并把结果存放在新的 list 中并返回。reduce()函数的参数和 map()一样，它计算元素的累加值，传入的函数的定义中必须具有两个参数。

这样一来，可以使用免费、轻量级的 Python 来取代付费、庞大的 MATLAB 进行科学计算和数据分析，而这只是 Python 强大的功能的一小部分

10.解线性方程组.py

用 scipy.linalg.solve(A,b)可以计算矩阵$A^{-1}b$，也就是解方程$\mathrm{Ax} = \mathrm{b}$

```
import scipy
from scipy import linalg
A = [[2, 1, 1, 4], [1, 2, -1, 4], [1, -1, 3, 3], [2, 1, -2, 2]]
b = [7, 5, 2, 9]
print scipy.linalg.solve(A, b)
```

运行结果如下：

```
[ 3.47619048 -0.23809524 -0.85714286  0.28571429]
```

像这样表示矩阵时，A 中第 3 个小的列表就是矩阵 A 的第 3 行

11.求矩阵的特征值、特征向量、对应行列式的值、逆.py

```
import scipy
from scipy import linalg
A = [[3, 0, -2, 1], [2, 1, 0, 1], [6, 3, 2, 5], [0, -1, -1, 2]]
B = [[-2, 1, 1], [0, 2, 0], [-4, 1, 3]]
print scipy.linalg.inv(A)
print scipy.linalg.det(A)
print scipy.linalg.eig(B)
```

运行结果如下：

```
    [[ 0.66666667 -1.75        0.41666667 -0.5       ]
 [-1.         3.75       -0.75        0.5       ]
 [ 0.33333333 -2.25        0.58333333 -0.5       ]
 [-0.33333333  0.75       -0.08333333  0.5       ]]
12.0
(array([-1.+0.j,  2.+0.j,  2.+0.j]), array([[-0.70710678, -0.24253563, 0.30151134],
       [ 0.        ,  0.        ,  0.90453403],
       [-0.70710678, -0.9701425 ,  0.30151134]]))
```

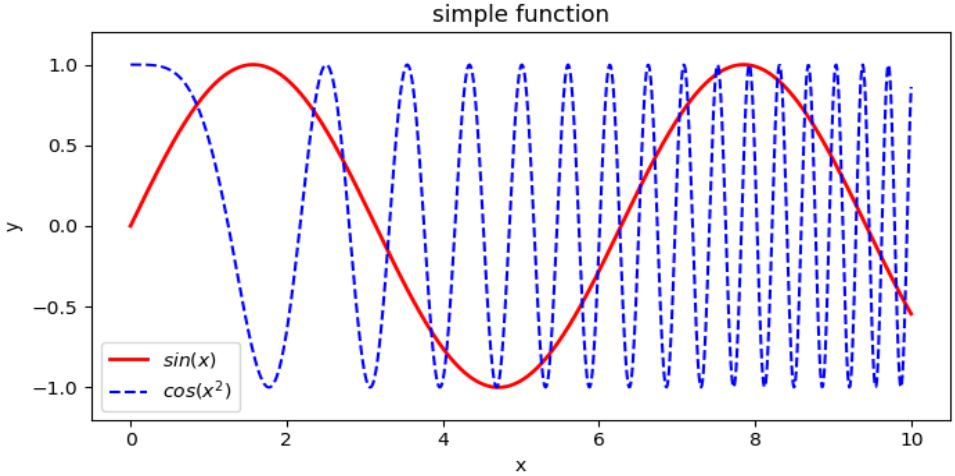
矩阵 B 的特征值为-1,2,2

12.求定积分.py

```
from scipy import integrate
import math
def f(x):
        return (math.cos(x) - (math.cos(x))**3)**0.5
e = integrate.quad(f, -(math.pi/2), (math.pi/2))
print e
```

运行结果如下：

```
(1.3333333333333328, 1.184918829721937e-11)
```

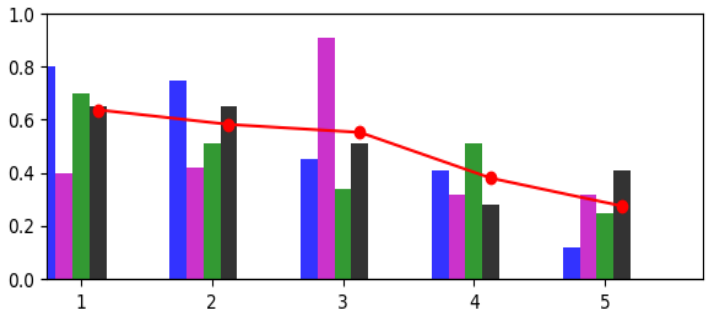


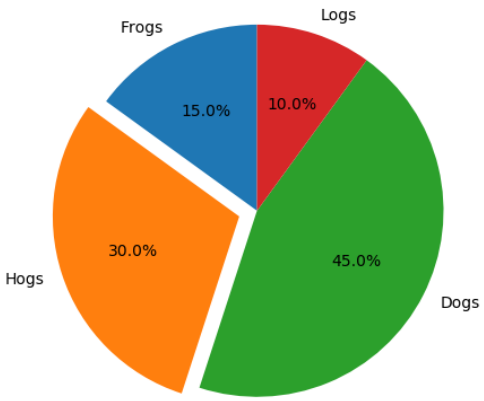
13.画简单二元函数的图像.py

```
import numpy as np
import matplotlib.pyplot as plt
x = np.linspace(0, 10, 1000)
y = np.sin(x)
z = np.cos(x**2)
plt.figure(figsize=(8, 4))
plt.plot(x, y, label="$sin(x)$", color="red", linewidth=2)
plt.plot(x, z, "b--", label="$cos(x^2)$")
plt.ylim(-1.2, 1.2)
plt.xlabel("x")
plt.ylabel("y")
plt.title("simple function")
plt.legend()
plt.show()
```

14.画饼形图.py

```
import matplotlib.pyplot as plt
labels = 'Frogs', 'Hogs', 'Dogs', 'Logs'
sizes = [15, 30, 45, 10]
explode = (0, 0.1, 0, 0)
plt.pie(sizes,   explode=explode,   labels=labels,   autopct='%1.1f%%',
shadow=False, startangle=90)
plt.axis('equal')
plt.show()         #
```





15.画竖直直方图.py

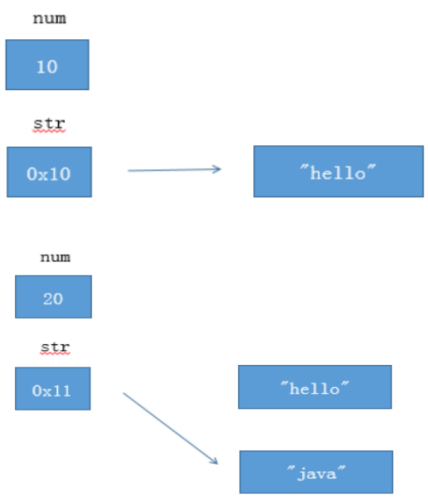
```
import numpy as np
from numpy import arange
import matplotlib.pyplot as plt
import scipy.io as sio
corr_m = [[0.8, 0.4, 0.7, 0.65], [0.75, 0.42, 0.51, 0.65], [0.45, 0.91, 0.34,
0.51], [0.41, 0.32, 0.51, 0.28], [0.12, 0.32, 0.25, 0.41]]
corr_m = np.array(corr_m)
mean_corr_m = np.mean(corr_m,axis=1)
N1, N2 = corr_m.shape
index = np.arange(N1) + 1
mean_corr_m_sort_location1 = np.argsort(mean_corr_m)
mean_corr_m_sort1 = np.sort(mean_corr_m)
mean_corr_m_sort_location = mean_corr_m_sort_location1[::-1]
mean_corr_m_sort = mean_corr_m_sort1[::-1]
corr_m_sort = np.zeros([5, 4])
for i in range(N1):
    corr_m_sort[i, :] = corr_m[mean_corr_m_sort_location[i], :]
fig = plt.figure(figsize=(6, 2.5))
ax = fig.add_subplot(111)
bar_width = 0.13
opacity = 0.8
rects1 = plt.bar(index, corr_m_sort[:, 0], bar_width, alpha=opacity,
color='b', label='Men')
rects2 = plt.bar(index+bar_width*1, corr_m_sort[:, 1], bar_width,
alpha=opacity, color='m')
rects3 = plt.bar(index+bar_width*2, corr_m_sort[:, 2], bar_width,
alpha=opacity, color='g')
rects4 = plt.bar(index+bar_width*3, corr_m_sort[:, 3], bar_width,
alpha=opacity, color='k')
plt.plot(index+bar_width*3, mean_corr_m_sort[:], 'ro-')
ax.set_xlim(1, 6)
ax.set_ylim(0, 1)
ax.set_xticks(np.linspace(1, 6, 6))
ax.set_yticks(np.linspace(0, 1, 6))
plt.xticks(index+2*bar_width, ('1', '2', '3', '4', '5'))
plt.tight_layout()
fig.set_size_inches(6, 2.5)
plt.show()
```

如果你没有弄清 C 语言中按值传递和按址传递的区别，暂且不管这个问题，看看 JAVA 是怎么做的：

首先要分清基本类型和引用类型

```
int num = 10;
String str = "hello";
```

num 是基本类型，值就直接保存在变量中；str 是引用类型，变量中保存的只是实际对象的地址，这种变量称为**引用**，它指向实际对象，实际对象中保存着内容

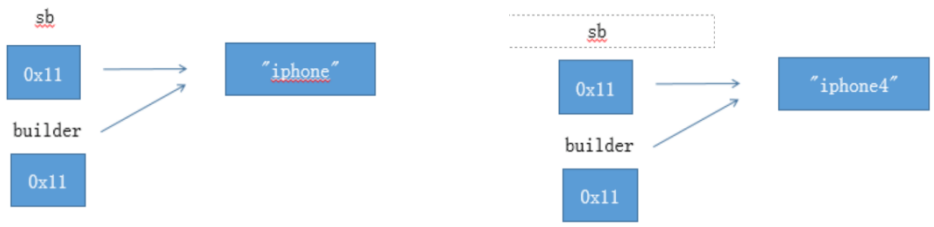



```
num = 20;
str = "java";
```

对于基本类型 num，赋值运算符会直接覆盖变量原来的值；对于引用类型 str，赋值运算符会改变引用中保存的地址，原来的地址被覆盖掉，但是原来的对象不会被改变(没有被任何引用所指向的对象是垃圾，会被垃圾回收器回收)

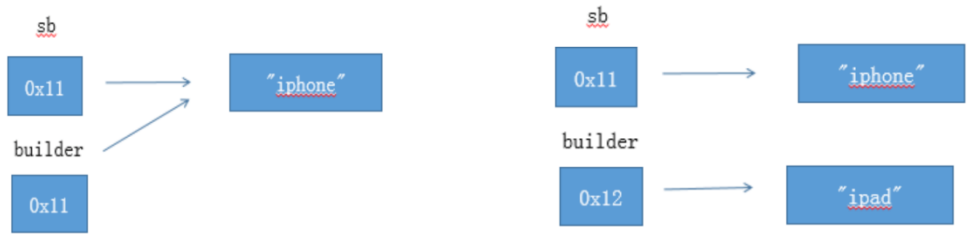
调用方法时发生了什么？JAVA 的参数传递基本上就是赋值操作！

```
StringBuilder sb = new StringBuilder("iphone");
void foo(StringBuilder builder){
    builder.append("4");                    //在这之后
}
foo(sb);
```



上面和下面的例子中都有引用类型 StringBuilder，这个类提供了改变自身的方法 append 而下面的例子没有用这个方法，而是用了赋值运算符，赋值只能让引用指向一个新的对象，而不会改变对象本身

```
StringBuilder sb = new StringBuilder("iphone");
void foo(StringBuilder builder){
    builder = new StringBuilder("ipad");      //在这之后
}
foo(sb);
```



你可以新建一个文本文件，然后把它的后缀名改成.csv，打开它你会发现它是一个简单的表格。这种文件在 win10 中的更改和保存有点繁琐，但是你只要一直点击“是”和“保存”就行了

比如说这样的一个 csv 文件：

1	Name	Physics	Python	Math	English
2	Google	37	123	299	47
3	Facebook	18	259	37	26
4	Twitter	23	479	69	89
5	Yahoo	12	45	57	90

把它和你的.py 源文件保存在同一个文件夹中，在.py 文件中编写以下代码：

```
import pandas as pd
Q = pd.read_csv("./Data2.csv")
print Q          #
```

输出结果是：

	Name	Physics	Python	Math	English
0	Google	37	123	299	47
1	Facebook	18	259	37	26
2	Twitter	23	479	69	89
3	Yahoo	12	45	57	90

Python 的数据分析库 pandas 中有两个特有的数据结构：Series 和 DataFrame
Series 相当于“竖着”的 list，每个数据对应一个索引 index(相当于其中一行)
DataFrame 是一种二维的数据表，它的列是 columns，行也是 index
上面的例子读取了一个 csv 文件并把 DataFrame 形式的数据赋值给了 Q

16.pandas 中两种数据结构的操作.py

```
import pandas as pd
from pandas import Series, DataFrame
s = Series([100, "Python", "what", "yes"])
print s
s1 = Series([100, "Python", "what", "yes"],
index=["mike", "title", "city", "lang"])
print s1
dic = {"C": 8000, "C++": 8500, "C#": 7900}
s2 = Series(dic)
print s2
s3 = Series(s2, index=["Java", "C", "Python",
"C#"])
print s3
print s3.isnull()
s4 = Series([3, 9, 4, 7], index=["a", "b", "c",
"d"])
print s4
print s4[s4 > 5]
print s4 * 5
data = {"name": ["yahoo", "google",
"facebook"], "marks": [200, 400, 800],
"price": [9, 3, 7]}
f1 = DataFrame(data, columns=['name',
'price' , 'marks', 'debt'], index=['a', 'b',
'c'])
print f1
f1['debt'] = 89.2
print f1
sdebt = Series([2.2, 3.3], index=["a", "c"])
f1['debt'] = sdebt
print f1
```

```
0       100
1    Python
2      what
3       yes
dtype: object
mike        100
title    Python
city       what
lang        yes
dtype: object
C       8000
C#      7900
C++     8500
dtype: int64
Java         NaN
C         8000.0
Python       NaN
C#        7900.0
dtype: float64
Java        True
C          False
Python      True
C#         False
dtype: bool
a    3
b    9
c    4
d    7
dtype: int64
b    9
d    7
dtype: int64
a    15
b    45
c    20
d    35
dtype: int64
       name  price  marks  debt
a     yahoo      9    200   NaN
b    google      3    400   NaN
c  facebook      7    800   NaN
       name  price  marks  debt
a     yahoo      9    200  89.2
b    google      3    400  89.2
c  facebook      7    800  89.2
       name  price  marks  debt
a     yahoo      9    200   2.2
b    google      3    400   NaN
c  facebook      7    800   3.3
```

图 G(V,E)的一条哈密顿回路是通过图中每个顶点的简单回路，具有这种回路的图称为哈密顿图，否则称为非哈密顿图。**哈密顿回路问题**：判断图 G 是否为哈密顿图

旅行商问题(TSP)与哈密顿圈问题有着密切的联系，在该问题中，一个售货员必须访问 n 个城市，如果把该问题抽象为一个具有 n 个顶点的完全图，就可以说这个售货员希望进行一次巡回旅行，可能会经过哈密顿回路，恰好访问每个城市一次，并最终回到出发的城市。从城市 i 到城市 j 的旅行费用为整数 c(i,j)，巡回的全部费用是旅行经过的各边的费用之和，而售货员希望使整个旅行费用最低。

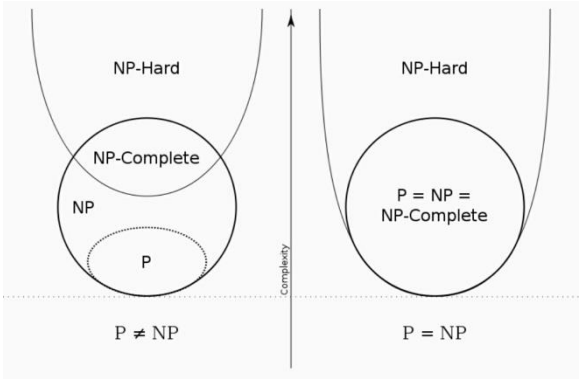
P 问题：一个问题可以在多项式时间内解决，也就是说，该问题具有确定的算法，并且算法的时间复杂度为 O(N^k)，其中 k 是一个常量

NP 问题：一个问题的某一个解可以在多项式时间内被证实或者证伪

NP-hard 问题：任何 NP 问题都可以在多项式时间内归约成这个问题，“归约”是一个转化的

过程，如果我可以把问题 A 中的一个实例转化为问题 B 的一个实例，然后通过解决问题 B 间接解决问题 A，那么就认为 B 比 A 难。原问题的答案和转化后问题的答案是相同的，我们解决了转化后的 NP-hard 问题，就解决了所有的 NP 问题

NP 完全问题(NPC 问题)：既是 NP 问题，又是 NP-hard 问题。这个定义可以推广到所有复杂度类，complete 的直观解释就是，我能解决这个问题就相当于具备了用相同级别的计算资源解决这个复杂度类里面所有问题的能力。



所有的 P 问题同时也是 NP 问题，因为如果一个问题是 P 问题，那么不需要任何证书(问题的一个已知的解)就能在多项式时间内解决它。

不太严谨的解释是：P 就是能在多项式时间内解决的问题，NP 就是能在多项式时间内验证答案是否正确的问题，**P=NP?**其实就是在问：如果对于一个问题我能在多项式时间内验证它的答案的正确性，那么我是否能在多项式时间内解决它？

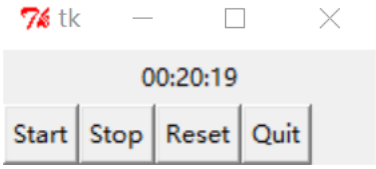
Python 自带了一个用于 GUI 编程的模块 Tkinter

17.在 Tkinter 中实现一个秒表.py

```
from Tkinter import *
import time
class StopWatch(Frame):
    msec = 50
    def __init__(self, parent=None, **kw):
        Frame.__init__(self, parent, kw)
        self._start = 0.0
        self._elapsedtime = 0.0
        self._running = False
        self.timestr = StringVar()
        self.makeWidgets()
    def makeWidgets(self):
        l = Label(self, textvariable=self.timestr)
        self._setTime(self._elapsedtime)
        l.pack(fill=X, expand=NO, pady=2, padx=2)
    def _update(self):
        self._elapsedtime = time.time() - self._start
        self._setTime(self._elapsedtime)
        self._timer = self.after(self.msec, self._update)
    def _setTime(self, elap):
        minutes = int(elap/60)
        seconds = int(elap - minutes*60.0)
        hseconds = int((elap - minutes*60.0 - seconds)*100)
        self.timestr.set('%02d:%02d:%02d' % (minutes, seconds, hseconds))
    def Start(self):
        if not self._running:
            self._start = time.time() - self._elapsedtime
            self._update()
            self._running = True
    def Stop(self):
        if self._running:
            self.after_cancel(self._timer)
            self._elapsedtime = time.time() - self._start
            self._setTime(self._elapsedtime)
            self._running = False
```

```
    def Reset(self):
        self._start = time.time()
        self._elapsedtime = 0.0
        self._setTime(self._elapsedtime)
if __name__ == '__main__':
    def main():
        root = Tk()
        sw = StopWatch(root)
        sw.pack(side=TOP)
        Button(root, text='Start', command=sw.Start).pack(side=LEFT)
        Button(root, text='Stop', command=sw.Stop).pack(side=LEFT)
        Button(root, text='Reset', command=sw.Reset).pack(side=LEFT)
        Button(root, text='Quit', command=root.quit).pack(side=LEFT)
        root.mainloop()
    main()
```

18.替换字符串中的子串.py



给定一个字符串，通过查询一个替换字典，将字符串中被标记的子串替换掉

```
def expand(format, d, marker='"', safe=False):
    if safe:
        def lookup(w): return d.get(w, w.join(marker*2))
    else:
        def lookup(w): return d[w]
    parts = format.split(marker)
    parts[1::2] = map(lookup, parts[1::2])
    return ''.join(parts)
if __name__ == '__main__':
    print expand('just "a" test', {'a': 'one'})
```

输出结果：just one test

对于大数运算，Python 是直接秒杀的，

计算 95.123 的 12 次方：

```
>>> a=95123**12
>>> b=3*12
>>> c='%d'%a
>>> d=c[0:-b]+'.'+c[-b:]
>>> print(d)
548815620517731830194541.899025343415715973535967221869852721
```

Python 是上限开区间，移小数点是从右往左

为了成为更加出色的程序员，必须深入了解计算机系统的工作原理，而不仅仅是会写代码而已。计算机系统是由硬件和系统软件组成的，我们编写的程序的生命周期是从源文件开始的，即使用文本编辑器创建并保存的文本文件。源程序实际上就是一个由值 0 和 1 组成的位(bit)序列，8 个位被组织成一组，称为字节，每个字节表示一个文本字符。

大部分的现代系统都使用 ASCII 标准来表示文本字符，这种方式就是用一个唯一的单字节大小的整数值来表示每个字符，例如换行符'\n'对应的整数值为 10。系统中的所有信息都是用一串位表示的，区分不同数据对象的唯一方法是我们读到这些数据对象时的上下文。

以 C 语言为例，在预处理阶段时，#include 命令告诉预处理器读取系统头文件 stdio.h 中的内容并把它直接插入到源程序中；然后是编译阶段，编译器把源程序翻译成汇编语言程序；接下来是汇编阶段，汇编程序被汇编器翻译成机器语言指令，这些指令被当做字节编码并打包存放在一个二进制文件 xxx.o 中；最后是链接阶段，链接器把单独的预编译好了的

目标文件(其中包含库函数)合并到 xxx.o 中，得到一个可执行文件，将被加载到**主存(内存)**中执行。
两种相关的机器语言是 **IA32**(32 位)和 **x86-64**(64 位)，计算机硬件能够直接执行的只有机器语言。
贯穿整个系统的是一组电子管道，叫做**总线**，它负责在各个部件之间传递信息
系统与外部世界的联系通道是**输入/输出(I/O)设备**，键盘和鼠标就是用户输入设备
内存是一个临时存储设备，在处理器执行程序时，用来存放程序和程序处理的数据。从物理上来说，内存是由一组动态随机存取存储器(DRAM)芯片组成的；从逻辑上看，存储器是一

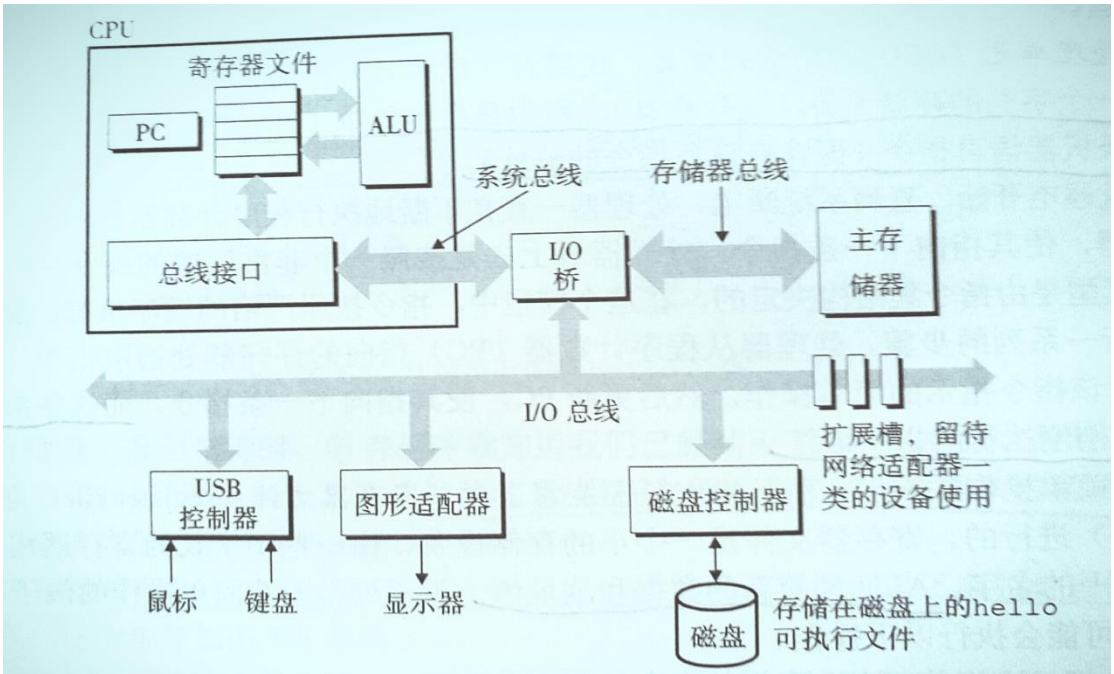
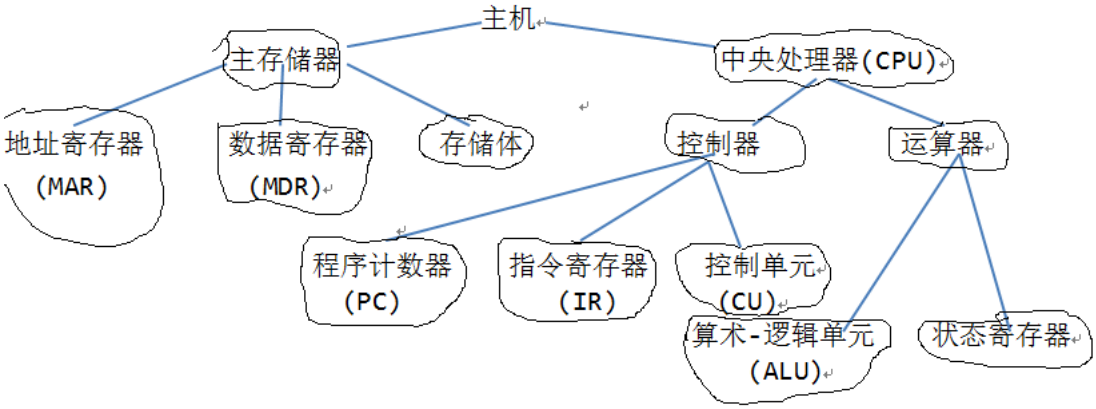


图 1-4　一个典型系统的硬件组成

CPU：中央处理单元；ALU：算术/逻辑单元；PC：程序计数器；USB：通用串行总线

个线性的字节数组，每个字节都有其唯一的地址，这些地址是从 0 开始的。



目前计算机中信息之所以用二进制表示，是因为这由物理器件的性能决定，0 和 1 正好对应逻辑电路中的低和高电平。
主存储器由许多存储单元组成，一个存储单元能存放一段位，这一段位的位数叫做**存储字长**，MAR 存放想要访问的存储单元的地址，它的位数等于存储单元的个数(**地址码长度**)；MDR 存放从存储单元取出来的数据，它的位数等于存储字长

IR 存放当前想要执行的指令，PC 存放下一条指令的地址。在任何时刻，PC 都指向主存中的某条机器语言指令，从系统通电到断电，CPU 一直在不断地执行 PC 指向的指令，再更新 PC，让它指向下一条指令

这样的现代计算机系统基于冯·诺依曼机，其核心思想是"存储程序"：采用控制流驱动方式，将指令以代码的形式事先输入到主存中，然后按一定的规则不断执行指令，直至结束

指令和数据以无法直接区分的方式一起放在内存中，它们都是用二进制，机器语言指令由操作码和地址码组成，操作码表示指令的性质或者作用，地址码用来告诉 CPU 在哪里能找到这条指令。一条指令的二进制代码的位数叫做**指令字长**，因为指令放在存储器中，所以指令字长一般是存储字长的整数倍，如果是 n 倍，取出这条指令就要访问内存 n 次。

CPU 不断地从内存中取出来东西，然后再判断取出来的是指令还是数据，如果是指令，就把指令的操作码送到 CU 那里去译码。至于 CPU 是怎么判断的，它是很巧妙的根据指令周期的不同阶段来判断的。

计算机能一次性直接处理的二进制代码的位数叫做**机器字长**，它一般等于 CPU 最内部的寄存器的大小，它决定了计算机的精度。

存储字长、指令字长、机器字长都必须是字节（8 位）的整数倍

计算机中的"透明"和政府说的"透明"是相反的，计算机中某个东西是透明的，意思是我们"看"不到这个东西。寄存器设置对汇编语言不透明，如果你用汇编语言编程，你必须知道(能直接看见)寄存器的结构和做法。但是有一种寄存器你看不见，它是 IR，它用来存放指令，你不需要看见它，你只需要移动指令(设置 PC)、知道程序的状态(看见状态寄存器)等。

一个 64 位的计算机用 16 位来表示地址，意思是说，机器字长=64，地址码长度=16，有2^{16}个地址空间。

和传统存储器不同的是，相联存储器访问内存时，既可按地址，也可按内容。

我们用这些词来作为计算机的性能指标：

吞吐量：　　系统在一秒内处理用户请求的数量，主要取决于内存的存取周期

响应时间：　　用户向计算机发送一个请求到计算机响应并且给出结果的等待时间

时钟周期：　　计算机操作的最小时间单位

CPU 周期：　　又叫机器周期，是多个时钟周期组成的

主频(CPU 时钟频率)：　时钟周期的倒数，一般以 MHZ(一百万赫兹)为单位

CPI：　　执行一条指令需要的时钟周期数

CPU 执行时间：　　运行一个程序所花费的时间

MIPS：　　一秒之内执行多少百万条指令

MFLOPS：　　一秒之内执行多少百万次浮点运算

CPU 执行时间 = 时钟周期数 * 时钟周期 = 时钟周期数/主频 = 指令条数 * CPI/主频

这个关系说明，CPU 性能取决于主频、CPI、指令条数

MIPS = 指令条数/(CPU 执行时间$\times 10^6$) = 主频/CPI

从上面的关系看出，主频不会影响到 CPI，但是能影响执行速度

根据机械原理，较大的存储设备要比较小的存储设备运行得慢，在处理器和一个又大又慢的设备之间插入一个更小更快的存储设备已经成为一个普遍的观念，实际上每个计算机系统中的存储设备都被组织成了一个存储器层次结构，在这个金字塔中，从塔顶到底部，设备的访问速度越来越慢，容量越来越大，并且每个字节的造价也越来越便宜。寄存器文件位于金字塔尖，也就是第 0 级或 L0，每一层上的存储器作为更低一级存储器的高速缓存(**Cache**)。如果你了解 Cache，你可以利用它极大地提高程序的性能

我们可以把操作系统看作是应用程序和硬件之间插入的一层软件，它有两个基本功能：1.防止硬件被失控的应用程序滥用，2.向应用程序提供一些简单一致的机制来控制复杂而又

通常大相径庭的低级硬件设备。操作系统通过 3 个抽象概念来实现这两个功能：**文件**是字节序列，**进程**是对一个正在运行的程序的一种抽象，**虚拟存储器**给进程提供了一个假象，即每个进程都在独占地使用内存。

从一个单独的系统来看，网络可视为一个 I/O 设备。

抽象是计算机科学中最为重要的概念之一，例如你不需要了解内部的实现就可以调用 API。在处理器里，指令集结构提供了对实际处理器硬件的抽象，使用这个抽象，机器代码的表现像是运行在一个一次只执行一条指令的处理器上，实际上底层硬件是并发执行的。

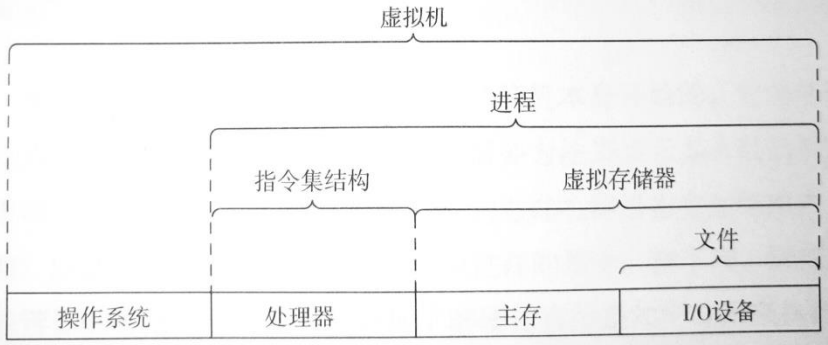


图 1-18　计算机系统提供的一些抽象

注：计算机系统中的一个重大主题就是提供不同层次的抽象表示，来隐藏实际实现的复杂性。

11. STL 的基本用法.Cpp

```
#include<cstdio>                            //STL:C++标准模板库
#include<vector>                            //向量(变长数组)
#include<set>      //集合(内部自动有序，不含重复元素的容器，红黑树实现)
#include<string>                            //字符串类
#include<map>           //建立两个基本类型之间的映射，红黑树实现
#include<queue>             //队列和优先队列(堆实现)
#include<stack>             //栈
#include<utility>
#include<algorithm>
using namespace std;
struct fruit                //水果结构体
{
    string name;
    int price;
    friend bool operator < (fruit f1,fruit f2)  //定义一个友元函数
    {                                   //C++不允许重载大于运算符
        return f1.price > f2.price;
    }       //把水果放进优先队列后，堆顶元素会是 price 最小的那一个水果
}f1,f2,f3;
bool cmp(string str1,string str2)       //用于 sort 的比较函数
{
    return str1.length()<str2.length();     //按长度从小到大排序
}
int main()
{
//vector
    vector<int> vi;                             //定义
```

```
    for(int i=0;i<=8;i++)
        vi.push_back(i);                //添加尾部元素
    for(vector<int>::iterator it=vi.begin();it!=vi.end();it++)
                                        //迭代器访问
        printf("%2d",*it);              //迭代器是一个类似指针的东西
    printf("\n%2d\n",vi.size());                    //打印元素个数
    vi.insert(vi.begin()+2,-1);                 //将-1 插入 vi[2]的位置
    vi.pop_back();                              //删除尾元素
    vi.erase(vi.begin()+4);                     //删除 vi[4]位置的元素
    vi.erase(vi.begin()+5,vi.begin()+7);   //删除 vi[5]到 vi[7-1]位置的元素
    for(int i=0;i<vi.size();i++)
        printf("%2d",vi[i]);                //下标访问
    vi.clear();                 //清空其中所有元素
    printf("\n");
//set
    set<int> st;
    for(int i=-2;i<=8;i+=2)
        st.insert(i);               //插入元素
    for(set<int>::iterator it=st.begin();it!=st.end();it++) //迭代器访问
        printf("%2d",*it);
    printf("\n%2d\n",st.size());
    st.erase(4);            //直接删除元素
    st.erase(st.find(8));           //用 find 函数找到迭代器，间接删除
    for(set<int>::iterator it=st.begin();it!=st.end();it++)
        printf("%2d",*it);
    st.clear();             //清空
    printf("\n");
//string
    string movie="The Dark Knight";
    string res="nice";
    movie.insert(3,res);        //在 movie[3]的位置插入 res
    printf("%s\n",movie.c_str());
                            //printf 不能直接打印 string 类的对象，要转换
    movie.erase(movie.begin()+13);      //删除 movie[13]位置的元素
    movie.erase(3,4);               //删除从 movie[3]开始的长度为 4 的一段字符
    printf("%s\n",movie.c_str());
    printf("%s\n",movie.substr(4,6).c_str());//从下标 4 开始的长度为 6 的子串
    if(movie.find("rk")!=string::npos)
        printf("%2d\n",movie.find("rk"));       //简单的模式匹配
    movie.replace(9,5,"princess"); //替换 movie[9]开始的长度为 5 的子串
    printf("%s\n",movie.c_str());
    movie.clear();
//map
    map<int,char> mp;                           //键:int,值:char,
    mp[4]='a';mp[2]='b';mp[7]='c';mp[5]='d';
    mp.erase(7);                                //删除键为 7 的一个映射
    for(map<int,char>::iterator mi=mp.begin();mi!=mp.end();mi++)
                                        //迭代器访问
        printf("%d %c\n",mi->first,mi->second);
                                        //first 表示键，second 表示值
    map<int,char>::iterator qs=mp.find(5); //用一个迭代器指向键为 5 的映射
    printf("%d %c\n",qs->first,qs->second);
    printf("%2d\n",mp.size());              //映射的对数
```

```
    mp.clear();
//queue
    queue<int> Q;                   //普通队列
    for(int i=1;i<=5;i++)
        Q.push(i);                  //依次入队
    printf("%2d%2d\n",Q.front(),Q.back());          //打印队头和队尾
    for(int i=1;i<=3;i++)
        Q.pop();                    //依次出队
    printf("%2d\n",Q.front());
    printf("%2d\n",Q.size());           //队中元素个数
    if(Q.empty()==true)
        printf("Empty\n");
    else
        printf("Not Empty\n");
    priority_queue<int> q1;                //默认为最大优先队列
      //定义一个最小优先队列，为了防止编译器把>>看做移位运算符，中间要有空格
    priority_queue<int,vector<int>,greater<int> > q2;
    q1.push(1);q1.push(5);q1.push(3);q1.push(4);
    q2.push(1);q2.push(5);q2.push(3);q2.push(4);
    printf("%2d%2d\n",q1.top(),q2.top());        //top 取优先级最高的元素
    for(int i=0;i<2;i++)
    {
        q1.pop();                       //队首(堆顶)元素出队
        q2.pop();
    }
    printf("%2d%2d\n",q1.top(),q2.top());

f1.name="apple";f1.price=19;f2.name="banana";f2.price=15;f3.name="orang
e";f3.price=17;
    priority_queue<fruit> q;
    q.push(f1);q.push(f2);q.push(f3);
    printf("%s\n",q.top().name.c_str());        //打印堆顶水果的名字
//stack
    stack<int> yes;
    for(int i=3;i<=9;i++)
        yes.push(i);           //压入
    for(int i=1;i<=4;i++)
        yes.pop();             //弹出
    printf("%2d%2d%2d\n",yes.top(),yes.size(),yes.empty());
                                  //栈顶元素，栈中元素个数，栈是否为空
//utility
    pair<string,int> p1,p2,p3;      //pair 用来构造一个单独的映射
    p1.first="haha";p1.second=5;    //3 种构造方法
    p2=make_pair("xixi",55);
    p3=pair<string,int>("heihei",555);
    map<string,int> good;
    good.insert(p1);good.insert(p2);good.insert(p3); //插入到 map 中
    for(map<string,int>::iterator mi=good.begin();mi!=good.end();mi++)
        printf("%s %d\n",mi->first.c_str(),mi->second);
//algorithm
    int x=1,y=6,z=3;
    printf("%2d%2d%2d\n",max(x,y),min(x,y),max(x,max(y,z)));
    swap(x,z);                                      //交换值
    printf("%2d%2d\n",x,z);
```

```
    int a[6]={1,2,3,4,5,6};
    reverse(a,a+4);                          //把 a[0]到 a[3]反转
    for(int i=0;i<6;i++)
        printf("%2d",a[i]);
    printf("\n");
    fill(a,a+3,0);                          //把 a[0]到 a[2]赋值为 0
    for(int i=0;i<6;i++)
        printf("%2d",a[i]);
    printf("\n");
    string alpha="abcdefghijk";
    reverse(alpha.begin()+2,alpha.begin()+6);
                                            //把 alpha[2]到 alpha[5]反转
    printf("%s\n",alpha.c_str());
    int b[10]={3,5,2,7,4,1,8,6,9,0};
    sort(b,b+6);                        //默认为把 b[0]到 b[5]从小到大排序
    for(int i=0;i<10;i++)
        printf("%2d",b[i]);
    printf("\n");
    string bad[3]={"aaa","bbbb","cc"};
    sort(bad,bad+3,cmp);                        //按字符串长度排序
    for(int i=0;i<3;i++)
        printf("%s\n",bad[i].c_str());
    return 0;
}
```

程序的运行结果是这样的：

```
 0 1 2 3 4 5 6 7 8
 9
 0 1-1 2 4 7
-2 0 2 4 6 8
 6
-2 0 2 6
Thenice Dark Knight
The Dark night
Dark n
 6
The Dark princess
2 b
4 a
5 d
5 d
 3
 1 5
 4
 2
Not Empty
 5 1
 3 4
banana
 5 3 0
haha 5
heihei 555
xixi 55
 6 1 6
 3 1
 4 3 2 1 5 6
 0 0 0 1 5 6
abfedcghijk
 1 2 3 4 5 7 8 6 9 0
cc
aaa
bbbb
```

这里有几个 Python 面试题答案：

```
# random 模块中的 shuffle(洗牌函数)
import random
alist = [1, 2, 3, 4]
random.shuffle(alist)
print alist
#list 和 tuple 之间的转化
#From list to Tuple
tuple(a_list)
#From Tuple to List
def to_list(t):
    return  [i  if  not  isinstance(i,tuple)  else
to_list(i) for i in t]
#tuple 中重复元素的出现次数
One=('a','l','u','a','t','t','a','p')
d={}
for k in One:
    for v in k:
        if v in d:
            d[v] += 1
        else:
            d[v] = 1
print d
#删除列表中的重复元素
a=[1,1,2,19,20,15,20]
c=sorted(set(a),key=a.index)
print c
```

19.求字符串中元音字母的个数.py

```
test_string = raw_input("enter a string:")
print len(filter(lambda x : x in 'aeiouAEIOU',test_string))
```

匿名函数用来检测字符是不是元音字母，作为 filter()函数的回调函数。filter()函数的功能是对一个可迭代对象中的每一个元素都用回调函数过滤并返回一个 list 对象。len()返回元素个数

Python 中用 and 、 not 、 or 表示逻辑关系运算符，用 elif 表示 else if

求出 i 的个、十、百位数的方法：a=i/1%10, b=i/10%10, c=i/100%10

62.花式输出字符串.C

```
#include<stdio.h>
typedef __int128_t dcf_t;
void face(dcf_t s,int i){
    int c=(((s&((dcf_t)0x1FULL<<i*5))>>i*5)+65);
    printf("%c",c);}
int main(void){
    for(int i=0;i<10;i++)
        face(0x28EC789572FC8 + ((dcf_t)0x000000<<64),i);}
```

运行结果：

I_LOVE_YOU

Python 中的 pandas 模块和 pylab 结合起来也可以画图，在 Series.plot()的参数中，kind 可以是’line’(折线图)、’bar’、(垂直柱状图)、’barh’、(水平柱状图)

random 中包含的随机数函数：

随机整数：randint(a,b)：返回随机整数 x,a<=x<=b

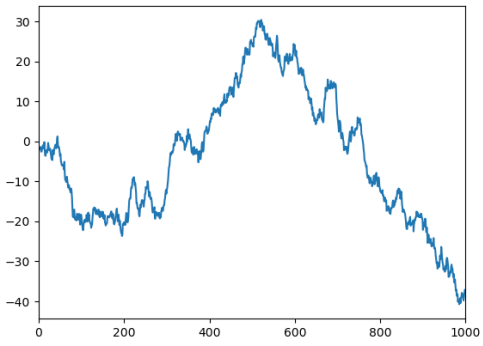
randrange(start,stop,[,step])：返回一个范围在(start,stop,step)之间的随机整数，不包括结束值。

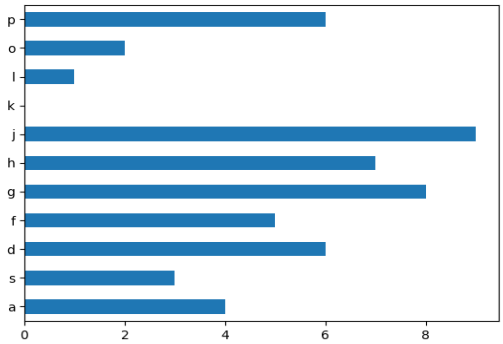
随机实数：random():返回 0 到 1 之间的浮点数

uniform(a,b):返回指定范围内的浮点数

randn(M):生成 M 个正态分布(平均值为 0，标准差为 1)的样本值

```
from pylab import *
import numpy as np
from pandas import Series
s = Series(np.random.randn(1000).cumsum(), index=np.arange(0, 1000))
s.plot(kind='line')
show()
```






```
from pylab import *
from pandas import Series,DataFrame
data=Series([4,3,6,5,8,7,9,0,1,2,6],index=list('asdfghjklop'))
data.plot(kind='barh')
show()
```

Pycharm 中的注释方法：Ctrl+/单行注释或取消注释

IDLE 中的注释方法：Alt+3 单行注释，Alt+4 取消注释

通用注释方法：用一对"""把你要注释的代码段括起来

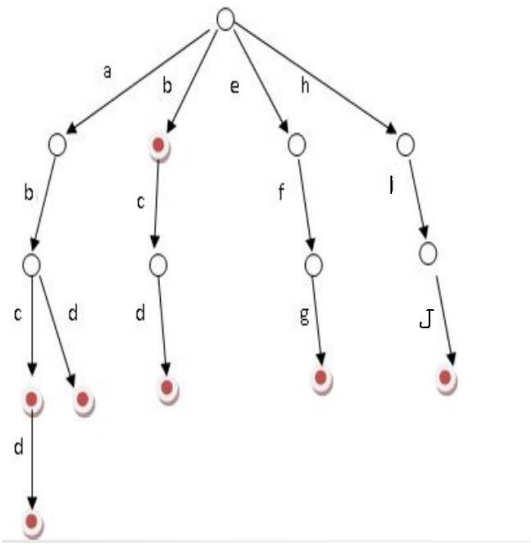
Python 中所有的东西都是对象。两种属性:实例属性、类属性
实例属性有两种定义方式：在类的外面显式定义、在类的里面通过构造函数__init__定义
类属性有两种：公有、私有(在属性名字前面加上两个下划线’__’就表示是私有的)
类有两种：旧式类、新式类(通过后面加上(object)来定义，它有一个独特的 super 函数)
方法有三种：对象方法(有一个 self 参数，类似 C++中的 this 指针)、静态方法(只能通过类的名字来调用，只能访问属于类的成员)、类方法(只能通过类的名字来调用，有一个 cls 参数，它和前者一样都要用修饰器@)

20.Python 面向对象的例子.py

```
#coding=utf-8
class that:
    word='yeah!'
    def say(self):
        print 'hello world'
print that.word
r=that()
r.say()
class fuck:
    name='jack'
p = fuck()
p.age=12
print p.name
print p.age
class shit:
    name='eson'
    __word='yeah'
    def __init__(self,age):
        self.age = age + 10
q=shit(12)
print q.name
print q.age
class my:
    def one(self):
        print 'method'
    @staticmethod                           #静态方法的修饰器
    def two():
        print 'static method'
    @classmethod
    def three(cls):
        print 'class method'
    def __say(self):                        #私有方法
        print 'there is nothing'
    def output(self):
        self.__say()
a=my()
a.output()
my.three()
class Complex:                              #定义复数类
    def __init__(self,real,image):
        self.real=real
        self.image=image
    def __add__(self, another):             #重载复数的加法运算符
        if type(another) is type(self):
            return
Complex(self.real+another.real,self.image+another.image)
```

```
        else:
            pass                                #跳过，什么也不做
x=Complex(3.0,-4.5)
y=Complex(2.0,5.5)
z=x+y
print z.real,z.image
class Person(object):                           #新式类
    def __init__(self):
        self.height = 160
    def about(self, name):
        print "{} is about {}".format(name, self.height)
class Girl(Person):                             #子类 Girl 继承自基类 Person
    def __init__(self):
        super(Girl, self).__init__()            #调用基类的方法，防止被覆盖
        self.breast = 90
    def about(self, name):
        print "{} is a hot girl, she is about {}, and her breast is
{}".format(name, self.height, self.breast)
        super(Girl, self).about(name)
if __name__ == "__main__":
    cang = Girl()
    cang.about("wangguniang")
```

```
yeah!
hello world
jack
12
eson
22
there is nothing
class method
5.0 1.0
wangguniang is a hot girl, she is about 160, and her breast is 90
wangguniang is about 160
```



由以下的串构成的一棵字典树：

ABC
ABCD
ABD
B
BCD
EFG
HIJ

求 1 到 10000 之间的所有完全数：

```
>>> [n for n in range(1,10000) if sum(k for k in range(1,n) if n%k==0)==n]
[6, 28, 496, 8128]
```

这里给一个 Python 数据分析的简单例子，它源于计算物理中将 res 文件转为 poscar 文件的一次不成熟的操作

```
TITL C
CELL    0.0000    2.4749    2.4749   17.5777   90.0000   90.0000  120.0000
LATT -1
SFAC As   C   O Te
C1     2   -0.00000    -0.00000     0.37067     1.00000     0.00000
C2     2    0.66667     0.33333     0.60059     1.00000     0.00000
C3     2    0.33333     0.66667     0.48563     1.00000     0.00000
TE4    4   -0.00000    -0.00000     0.62933     1.00000     0.00000
AS5    1    0.66667     0.33333     0.51437     1.00000     0.00000
TE6    4    0.33333     0.66667     0.39941     1.00000     0.00000
O7     3    0.49505    -0.53277    -0.30151     1.00000     0.00000
```

上面是要被转换的文本文件 res.txt

21.计算物理问题:res 文件转 poscar 文件.py

```
import pandas as pd
import numpy as np
from math import*
from collections import Counter
from pandas import DataFrame
np.set_printoptions(suppress=True, formatter={'float': '{:0.3f}'.format})
data = pd.read_table('res.txt', skiprows=[0, 2, 3], header=None, delim_whitespace=True, error_bad_lines=False)
f1 = DataFrame(data)
s1 = np.array(f1)
length = s1[0, 2:5]
q = s1[0, 5:]
angle = [radians(q[0]), radians(q[1]), radians(q[2])]
k1 = [length[0], 0, 0]
k2 = [length[1]*cos(angle[2]), length[1]*sin(angle[2]), 0]
k3 = [length[2]*cos(angle[1]), length[2]*(cos(angle[0])-cos(angle[1])*cos(angle[2])/sin(angle[2])), length[2]*(sqrt(1 + 2*cos(angle[0])*cos(angle[1])*cos(angle[2])-(cos(angle[0]))**2-(cos(angle[1]))**2-(cos(angle[2]))**2)/sin(angle[2]))]
k = np.array([k1, k2, k3])
Aname = pd.read_table('res.txt', skiprows=range(0, 3)+range(4, 11), header=None, delim_whitespace=True)
temp = np.array(Aname)
h1 = temp[0, 1:]
atoms = s1[1:, 0]
number = (Counter([i[0] for i in atoms])).values()
vector = s1[1:, 2:5]
with open('POSC_.txt', 'w') as f:
    f.write("System ")
    f.write(str(h1)+'\n')
    f.write(str(k))
    f.write('\n'*2)
    f.write(str(number))
    f.write('\n'+'direct'+'\n')
    f.write(str(vector))
with open('POSC_.txt', 'r') as fpr:
    content = fpr.read()
content = content.replace('[', '')
```

```
content = content.replace(']', '')
content = content.replace(',', '')
content = content.replace("'", '')
with open('POSC_.txt', 'w') as fpw:
    fpw.write(content)
```

这是程序运行后生成的 POSC_.txt

```
System As C O Te
2.475 0.000 0.000
 -1.237 2.143 0.000
 0.000 0.000 17.578

1 3 2 1
direct
-0.0 -0.0 0.37067
 0.66667 0.33333 0.60059
 0.33333 0.66667 0.48563
 -0.0 -0.0 0.6293300000000001
 0.66667 0.33333 0.51437
 0.33333 0.66667 0.39941
 0.49505 -0.53277 -0.30151
```

Python 中的类都具有一些由”__”开头结尾的固定方法，称为魔术方法，例如__add__

广播是指不同形状的数组之间的算术运算的执行方式，它是一种非常强大的功能。广播的原则是:如果两个数组的后缘维度(从末尾开始算起的维度)的轴长相符或其中一个的长度为 1，则认为它们是广播兼容的，广播会在缺失和长度为 1 的维度上进行

在 numpy 中可以用 .T 来转置矩阵，c = np.arange(15).reshape(5, 3)将生成一个 5 行 3 列的矩阵，其中的元素是 0 到 14 并按行递增。

63.打印一个占据 19 行的空心菱形.C

```
main(_){for(_=0;_++<381;)printf(_%20?abs(_%20-10)+abs(_/20-9)-9?"
":"*":"\n");}
```

在声明变量、函数、和大多数其他类型的实体的时候，C++要求使用指定的类型。然而对于许多代码，除了类型不同之外，其余的代码看起来都是相同的。**模板(Templates)**是一些为多种类型而编写的函数和类，而且这些类型都没有指定。当使用模板时你只需要把所希望的类型作为一个(显式或者隐式的)实际参数传递给模板。

12.返回两个数中较大者的函数模板.Cpp

```
#include<iostream>
#include<string>
template <typename T> inline T const &bigger(T const &a,T const &b){
    return a<b?b:a;
}
int main(){
    int i=8;double f1=3.4,f2=-6.7;std::string s1="mathematics",s2="math";
    std::cout<<bigger(7,i)<<bigger(f1,f2)<<bigger(s1,s2)<<std::endl;
}
```

运行结果：

83.4mathematics

函数模板有两种参数：

```
template<typename T1,typename T2>                //T1,T2 是模板参数
inline T1 bigger(T1 const &a,T2 const &b){       //a,b 是调用参数
    return a < b ? b : a;          //OK,但第一个模板实参的类型定义了返回类型
}
```

模板参数可以通过我们所传递的实参来决定(实参演绎)，也可以显式的实例化：

bigger<double>(4,4.2);

类模板是具有如下性质的类:在类的实现中，可以有一个或多个类型还没有被指定。为了使用它，你可以传入某个具体类型作为模板实参。对于类模板，只有那些被调用的成员函数才会被实例化。为了定义类模板的成员函数，你必须指定该成员函数是一个函数模板，而且你还要使用这个类模板的完整类型限定符

Partial template specialization is a particular form of class [template specialization](). it allows the programmer to specialize only some arguments

of a class template, as opposed to explicit specialization, where all the template arguments are provided

这里有一个用 C++标准库的类模板 vector<>实现的简单的类模板 Stack<>

```
#include<vector>
#include<stdexcept>
template<typename T>
class Stack{
private:
    std::vector<T> elems;                    //存储元素的容器
public:
    void push(T const&);                     //压入元素
    T top() const;                           //返回栈顶元素
    bool empty() const{                      //判断栈是否为空
        return elems.empty();
    }
};
template<typename T> void Stack<T>::push(T const &elem){
    elems.push_back(elem);              //把 elem 的拷贝附加到末尾
}
template<typename T> T Stack<T>::top() const{
    if(elems.empty()){                          //捕获并抛出异常
        throw std::out_of_range("Stack<>::pop(): empty stack");
    }
    return elems.back();         //返回最后一个元素的拷贝
}
int main(void){return 0;}
```

红黑树是一种二叉查找树，它的性质是：每个节点要么是红的要么是黑的；根节点和叶子节点一定是黑的；红色节点的两个孩子节点都是黑的；对每一个节点，从它到它的所有后代叶子节点的简单路径上，均包含相同数目的黑色节点；它有一个黑色的边界哨兵 NIL 用来收集根节点和叶子节点的 NULL 指针

```
#include<iostream>
template<typename T> class RBTnode{     //红黑树节点
    //friend class RBT<Datatype>;
public:
    RBTnode(){                           //constructor:构造 NIL 节点
        lChild=Rchild=parent=NULL;
        color=0;                         //黑色的
        data=NULL;
    }
    RBTnode(T d,int color){                //构造内节点
        data=d;
        lChild=Rchild=parent=NULL;
        this->color=color;
    }
    ~RBTnode(){                           //destructor
        lChild=Rchild=parent=NULL;
        delete this;
    }
private:
    RBTnode<T> *lChild,*Rchild,*parent;     //指针域
    T data;                              //数据域
    int color;                        //颜色域，0 是黑色，1 是红色
};
```

```
template<typename T> class RBT{             //红黑树
public:
    RBT(){                                  //constructor
        root=NULL;
        NIL=new RBTnode<T>();
        count=0;
    }
    RBT(T data){
        root=new RBTnode<T>(data,0);            //创建黑色的根节点
        NIL=new RBTnode<T>();
        count=1;
    }
    void LeftRotated(RBTnode<T> *node);         //左旋转
private:
    RBTnode<T> *root;                           //根节点
    RBTnode<T> *NIL;                            //哨兵
    int count;                                  //节点数目
};
int main(void){return 0;}
```

C++中，如果某个类的构造函数只有一个参数，那么在给该类对象赋值时就会有一个缺省的转换操作，例如 CxString string2=10;这段代码，编译器会自动把那一个参数(10)的类型(整型)转换成 CxString 类对象，这样显得不伦不类，解决方法就是在构造函数前面加上 explicit 关键字，它的作用是防止 constructor 的隐式类型转换，这种情况下只能改写成 CxString string(10);(显式调用)

C++中有 4 种强制类型转换：

static_cast<double>(4)　可以实现内置基本数据类型之间的转换，如果涉及到类，那么这两个类必须要有某种相互联系，这种联系不一定包含虚函数

const_cast　仅仅把一个它作用的表达式转换为常量，否则反过来，把 const 属性去掉

reinterpret_cast　具有和 C 风格一样的各种类型之间转换的能力，不到万不得已不要随便使用

dynamic_cast　程序运行时转换，不能用于内置基本数据类型，返回的是指向类的指针或引用，使用它来进行转换的，基类中一定要有虚函数，否则编译不过

模板元编程(template metaprogramming)是一种随着模板实例化所完成的编译期计算，之所以能这样实现，是因为模板实例化机制是一种基本的递归语言机制

13.计算点乘.Cpp

```
#include<iostream>
template<int DIM,typename T> class DotProduct{      //基本模板
    public:
    static T result(T* a,T* b){
        return *a * *b + DotProduct<DIM-1,T>::result(a+1,b+1);
    }
};
template<typename T> class DotProduct<1,T>{     //作为结束条件的局部特化
    public:
    static T result(T* a,T* b){
        return *a * *b;
    }
};
template<int DIM,typename T> inline T dot(T *a,T* b){   //辅助函数
    return DotProduct<DIM,T>::result(a,b);
}
```

```
int main(){
    int a[3]={1,2,3};int b[3]={5,6,7};
    std::cout<<dot<3>(a,b);                //输出 38
}
```

输入是运行时变量，所以元编程一般不能支持用户输入

元编程侧重于用代码生成代码，泛型编程侧重于减少代码对特定的数据类型的依赖，这两件事 C++ Template 都可以做

学过 STL 之后，做算法题就不需要自己实现基本数据结构了

14.简单计算器.Cpp

```
#include<iostream>
#include<cstdio>
#include<string>
#include<stack>
#include<queue>
#include<map>
using namespace std;
struct node{
    double num;                    //操作数
    char op;                       //操作符
    bool flag;                     //true 表示数，false 表示符
};
string str;
stack<node> s;                         //操作符栈
queue<node> q;                         //后缀表达式序列
map<char,int> op;
void change(){                         //中缀表达式转后缀表达式
    double num;
    node temp;
    for(int i=0;i<str.length();){
        if(str[i]>='0'&&str[i]<='9'){          //如果是数字
            temp.flag=true;
            temp.num=str[i++]-'0';     //记录这个操作数的第一个数位
            while(i<str.length()&&str[i]>='0'&&str[i]<='9'){
                temp.num=temp.num*10+(str[i]-'0'); //更新这个操作数
                i++;
            }
            q.push(temp);       //把这个数放入后缀表达式的队列
        }
        else{                      //如果是操作符
            temp.flag=false;
            while(!s.empty()&&op[str[i]]<=op[s.top().op]){
                q.push(s.top());
                s.pop();
            }
            temp.op=str[i];
            s.push(temp);              //把这个操作符压入操作符栈中
            i++;
        }
    }
    while(!s.empty()){             //如果操作符栈中还有操作符
        q.push(s.top());           //弹出
        s.pop();                   //到后缀表达式队列中
    }
```

```
}
double Cal(){                        //计算后缀表达式
    double temp1,temp2;
    node cur,temp;
    while(!q.empty()){          //只要后缀表达式队列不空
        cur=q.front();                  //cur 记录队首元素
        q.pop();
        if(cur.flag==true) s.push(cur);   //如果是操作数，直接压入栈
        else{                       //如果是操作符
            temp2=s.top().num;              //弹出第二操作数
            s.pop();
            temp1=s.top().num;              //弹出第一操作数
            s.pop();
            temp.flag=true;                 //临时记录操作数
            if(cur.op=='+') temp.num=temp1+temp2;
            else if(cur.op=='-') temp.num=temp1-temp2;
            else if(cur.op=='*') temp.num=temp1*temp2;
            else temp.num=temp1/temp2;
            s.push(temp);                   //把这个操作数压入栈
        }
    }
    return s.top().num;         //栈顶元素就是后缀表达式的值
}
int main(){
    op['+']=op['-']=1;                  //设定操作符的优先级
    op['*']=op['/']=2;
    while(getline(cin,str),str!="0"){
        for(string::iterator it=str.end();it!=str.begin();it--){
            if(*it==' ') str.erase(it);     //去掉表达式中的空格
        }
        while(!s.empty()) s.pop();      //初始化栈
        change();
        printf("%.3f\n",Cal());
    }
    return 0;
}
```

15.已知二叉树的后序、中序遍历序列，求层序遍历序列.Cpp

```
#include<cstdio>
#include<cstring>
#include<queue>
#include<algorithm>
using namespace std;
const int maxn=50;
struct node{
    int data;
    node *lchild;
    node *rchild;
};
int pre[maxn],in[maxn],post[maxn];          //先序、中序、后序
int n;                                      //结点个数
//假设递归过程中某步的后序序列区间为[postL,postR]，中序序列区间为[inL,inR]
node* create(int postL,int postR,int inL,int inR){  //返回重建的树的根结点
    if(postL>postR) return NULL;    //后序序列长度不大于 0 时直接返回
    node *root=new node;
```

```
    root->data=post[postR];     //后序序列的最后一个元素就是根结点
    int k;
    for(k=inL;k<=inR;k++){
        if(in[k]==post[postR])      //在中序序列中找到根结点
            break;
    }
    int numLeft=k-inL;          //左子树的结点个数
    root->lchild=create(postL,postL+numLeft-1,inL,k-1);
    root->rchild=create(postL+numLeft,postR-1,k+1,inR);
    return root;
}
int num=0;                  //已经输出的结点个数
void BFS(node *root){
    queue<node*> q;                 //注意队列里面是存地址
    q.push(root);                   //根结点地址入队
    while(!q.empty()){
        node *now=q.front();            //取出队首元素
        q.pop();
        printf("%d",now->data);             //访问队首元素
        num++;
        if(num<n) printf(" ");
        if(now->lchild!=NULL) q.push(now->lchild);
        if(now->rchild!=NULL) q.push(now->rchild);
    }
}
int main(){
    scanf("%d",&n);
    for(int i=0;i<n;i++)
        scanf("%d",&post[i]);
    for(int i=0;i<n;i++)
        scanf("%d",&in[i]);
    node* root=create(0,n-1,0,n-1);     //建树
    BFS(root);                          //层次遍历
    return 0;
}
```

输入样例：

7

2 3 1 5 7 6 4

1 2 3 4 5 6 7

输出样例：

4 1 6 3 5 7 2

16.求树中权重相同的路径.Cpp

给定一棵树和每个结点的权重，求所有从根结点到叶子结点的路径，使得每条路径上的结点的权重之和等于给定的常数 S，如果有多条这样的路径，则按路径非递增的顺序输出

输入样例：

20 9 24

10 2 4 3 5 10 2 18 9 7 2 2 1 3 12 1 8 6 2 2

00 4 01 02 03 04

02 1 05

04 2 06 07

03 3 11 12 13

06 1 09

07 2 08 10

16 1 15
13 3 14 16 17
17 2 18 19
输出样例：
10 5 2 7
10 4 10
10 3 3 6 2
10 3 3 6 2

```
#include<cstdio>
#include<vector>
#include<algorithm>
using namespace std;
const int MAXN=110;
struct node{
    int weight;                     //数据域
    vector<int> child;              //指针域
}Node[MAXN];                        //结点数组
bool cmp(int a,int b){              //按结点数据域从大到小排序
    return Node[a].weight > Node[b].weight;
}
int n,m,S;                          //结点数、边数、给定的和
int path[MAXN];                     //记录路径
//当前访问结点为 index，numNode 为当前路径 path 上的结点个数，sum 为当前的结点权重之和
void DFS(int index,int numNode,int sum){
    if(sum>S) return;
    if(sum==S){
        if(Node[index].child.size()!=0) return;     //还没到叶子结点
        for(int i=0;i<numNode;i++){     //到达叶子结点，输出路径
            printf("%d",Node[path[i]].weight);
            if(i<numNode-1) printf(" ");
            else printf("\n");
        }
        return;
    }
    for(int i=0;i<Node[index].child.size();i++){    //枚举所有子结点
        int child=Node[index].child[i];     //结点 index 的第 i 个子结点编号
        path[numNode]=child;        //把结点 child 加到路径 path 末尾
        DFS(child,numNode+1,sum+Node[child].weight); //递归进入下一层
    }
}
int main(){
    scanf("%d%d%d",&n,&m,&S);
    for(int i=0;i<n;i++)
        scanf("%d",&Node[i].weight);
    int id,k,child;
    for(int i=0;i<m;i++){
        scanf("%d%d",&id,&k);               //结点编号、孩子个数
        for(int j=0;j<k;j++){
            scanf("%d",&child);
            Node[id].child.push_back(child);    //child 为结点 id 的孩子
        }
        sort(Node[id].child.begin(),Node[id].child.end(),cmp);
    }
```

```
    path[0]=0;              //路径的第 1 个结点设置为 0 号结点
    DFS(0,1,Node[0].weight);
    return 0;
}
```

记忆化搜索：将已经计算过的内容记录下来，下一次计算时直接调用上一次的结果

例如求解斐波那契数列时，用一个 O(N)的空间复杂度把 O(2^N)的时间复杂度降到 O(N)

动态规划(Dynamic Programming,DP)是一种用来解决一类最优化问题的算法思想

```
#include<algorithm>
using namespace std;
int Fib(int n){
    const int MAXN=100;
    int dp[MAXN];
    fill(dp,dp+MAXN,-1);
    if(n==0||n==1)
        return 1;
    if(dp[n]!=-1)
        return dp[n];
    else{
        dp[n]=Fib(n-1)+Fib(n-2);
        return dp[n];
    }
}
```

如果一个问题可以被分解为若干个子问题，且这些子问题会重复出现，那么就说这个问题拥有重叠子问题

如果一个问题的最优解可以由其子问题的最优解有效地构造出来，那么就说这个问题具有最优子结构

一个问题必须拥有重叠子问题和最优子结构，才能使用动态规划去解。通过记录重叠子问题的解，使下次碰到相同的子问题时直接使用之前记录的结果，从而避免重复计算；最优子结构保证原问题的最优解可以由子问题的最优解推导而来

动态规划的核心在于设计状态并设计表示最优子结构的推导关系的状态转移方程

22.计算物理问题:poscar 文件转 res 文件.py

```
SYSTEM=HgTe
   1.00000000000000
     8.1102000000000007    0.0000000000000000    0.0000000000000000
     0.0000000000000000    4.6037885379335197    0.0000000000000000
     0.0000000000000000    0.0000000000000000   19.9845725755607937
   Hg   Te
     4     4
Direct
  0.9975082190336053  0.0000000000000000  0.4454738417796165
  0.4975082190336053  0.5000000000000000  0.4454738417796165
  0.4975082190336053  0.0000000000000000  0.5545261582203835
  0.9975082190336053  0.5000000000000000  0.5545261582203835
  0.1691617809663981  0.5000000000000000  0.4048887889157271
  0.6691617809663981  0.0000000000000000  0.4048887889157271
  0.1691617809663981  0.0000000000000000  0.5951112110842729
  0.6691617809663981  0.5000000000000000  0.5951112110842729

  0.00000000E+00  0.00000000E+00  0.00000000E+00
  0.00000000E+00  0.00000000E+00  0.00000000E+00
  0.00000000E+00  0.00000000E+00  0.00000000E+00
  0.00000000E+00  0.00000000E+00  0.00000000E+00
```

```
import pandas as pd
from pandas import DataFrame
```

```
from math import*
import numpy as np
np.set_printoptions(suppress=True, formatter={'float': '{:0.5f}'.format})
pd.set_option('precision', 5)                                    #set precision
data = pd.read_table('CONTCAR', sep=' ')                      #read it simply
f1 = DataFrame(data)
s1 = np.array(f1)
K = [0.0, 0.0, 0.0]
for i in range(0, 3):                                         #skipover Nan values
    K[i] = np.array([x for x in s1[i+1] if str(x) != 'nan'], dtype=float)
K = np.array(K)
f = lambda q: q*180/pi                                       #transfer rad to angle
angle    =    [f(acos(np.vdot(K[0],    K[1]))),    f(acos(np.vdot(K[0],    K[2]))),
f(acos(np.vdot(K[1], K[2])))]
length   =   np.array([sqrt(np.vdot(K[0],   K[0])),   sqrt(np.vdot(K[1],   K[1])),
sqrt(np.vdot(K[2], K[2]))])
Aname = [x for x in s1[4] if str(x) != 'nan']
number = [x for x in s1[5] if str(x) != 'nan']
number = np.array(number, dtype=int)
g = sum(number)                                                  #number of atoms
kh = []
for i in range(0, g):
    kh.append(0.0)
for i in range(0, g):
    kh[i] = [x for x in s1[i+g-1] if str(x) != 'nan']
kh = np.array(kh, dtype=float)
w1 = np.ones(g)
w2 = np.zeros(g)
kh = np.insert(kh, 3, values=w1, axis=1)           #insert a column into Dataframe
kh = np.insert(kh, 4, values=w2, axis=1)
r1 = []
for i in range(0, 5):
    r1.append('%')
Atom = []
Atype = []
for k in range(0, len(Aname)):
    for i in range(0, number[k]):
        Atom.append(Aname[k])
        Atype.append(k+1)
for i in range(0, g):
    Atom[i] += str(i+1)
kh = DataFrame(kh, index=Atom, columns=r1, dtype=float)
kh.insert(0, '(', Atype, allow_duplicates=True)
e1 = 'CELL   0.0000   '
with open('temp.txt', 'w') as f:
    f.write('TITL C'+'\n')
    f.write(e1)
    f.write(str(length)+'  ')
    f.write(str(angle)+'\n')
    f.write('LATT -1'+'\n')
    f.write('SFAC  ')
    f.write(str(Aname)+'\n')
    f.write(str(kh)+'\n')
    f.write('END')
with open('temp.txt', 'r') as fpr:
    content = fpr.read()
    for i in ['[', ']', ',', "'", '%', '(']:            #delete these in temp.txt
        content = content.replace(i, '')
with open('temp.txt', 'w') as fpw:
    fpw.write(content)
def delblankline(infile, outfile):                   # Delete blanklines of infile
    infp = open(infile, "r")
    outfp = open(outfile, "w")
    lines = infp.readlines()
    for li in lines:
```

```
        if li.split():
            outfp.writelines(li)
    infp.close()
    outfp.close()
delblankline("temp.txt", "HAHA.txt")
```

```
TITL C
CELL   0.0000   8.11020 4.60379 19.98457  90.0 90.0 90.0
LATT -1
SFAC  Hg Te
Hg1  1  0.99751  0.0  0.44547  1.0  0.0
Hg2  1  0.49751  0.5  0.44547  1.0  0.0
Hg3  1  0.49751  0.0  0.55453  1.0  0.0
Hg4  1  0.99751  0.5  0.55453  1.0  0.0
Te5  2  0.16916  0.5  0.40489  1.0  0.0
Te6  2  0.66916  0.0  0.40489  1.0  0.0
Te7  2  0.16916  0.0  0.59511  1.0  0.0
Te8  2  0.66916  0.5  0.59511  1.0  0.0
END
```

改变元组的方法：

```
In [1]: a=(3,'k',8.99)
In [2]: b=(0,)
In [3]: c=a.__add__(b)
In [4]: c
Out[4]: (3, 'k', 8.99, 0)
```

root 是 Linux 系统中万能的天神

linux 系统是区分大小写的

df -h 列出文件系统的整体磁盘使用量

cd ..　　返回上级文件目录

cd　　回到当前文件夹

ls　　显示文件夹中文件

ls -l POSCAR_example　　显示文件 POSCAR_example 的最近修改时间

bc　　简单计算器

scale=3　　设置计算器显示小数点后 3 位

quit　离开计算器

date　　显示日期

cal　　显示本月的日历

cal 7 2099　　显示任意年月的日历

cal 2017　　显示全年的日历

tab 键可以用于补全文件名和命令名，

Ctrl+c　　将正在运行中的命令中断

man ls　　获得命令 ls 的帮助信息，按 q 退出

ls -al　　显示文件的各种属性

mkdir this　　在当前文件夹中新建一个 this 文件夹

rmdir this　　删除它　　rm　　删除一个文件

cat　　直接查看文件内容

cat -n　　打印行号，包括空白行的行号

head -4 CONTCAR　　显示文件“CONTCAR”的前 4 行

tail -n +5 CONTCAR　　列出该文件第 5 行之后(包括第 5 行)的全部内容

tail -3 CONTCAR　　列出文件的最后 3 行

head -n 15 CONTCAR | tail -n 10　　列出文件第 6 行到第 15 行

grep -n "\0.0000000000000000" CONTCAR　　显示包含特定字符串的文件中的行以及行号

vim hahaaaa.txt　　新建一个文本文件，按 i 进入插入的编辑模式，按 esc 退出插入模式，再按:将光标移动到最下面，按 wq 保存文件到当前文件夹并退出。之后再次使用这个命令时：

移动光标：　0 移动到这一行的最前面字符　$ 移动到这一行的最后面字符处　G 移动到该

文件的最后一行　gg 移动到第一行
查找与替换：　:1,$s/2/O/g　在整个文件中查找字符“2”并将其替换为字符“0”　:5,7s/0/1/g　在第 5 行和第 7 行之间(包括第 5 行和第 7 行)查找字符“0”并把它替换为字符“1”
dd　删除光标所在那一行
不能使用 backspace 键来删除字符，x　向后删除一个字符　X 向前删除一个字符(相当于 backspace)
yy 复制光标所在的那一行　p 将已经复制的数据在光标下一行粘贴
9yy 复制光标所在的向下 9 行
小数点.　重复前一个操作
按下 R 进入替换模式，会一直替换光标所在的字符，直到按下 esc 为止
按 u 可以回退并撤销之前的操作
管道命令：加入“|”，它把两个命令分开并连续执行，后面的命令接受前面的命令的输出
vim 可以编写程序源文件，具有代码高亮功能，在 Linux 下运行 helloworld 程序：
[student@node13 ~]$ gcc helloworld.c
[student@node13 ~]$ ./a.out
hello world
在 Linux 系统中运行 Python 脚本：[student@node13 ~]$ python this.py
0 1 2 3 4 5 6 7
在计算机科学中，Shell 俗称壳（用来区别于核），是指“提供使用者使用界面”的软件（命令解释器）。它类似于 DOS 下的 command 和后来的 cmd.exe。它接收用户命令，然后调用相应的应用程序
bash 是大多数 Linux 系统以及 Mac OS X 默认的 shell，它能运行于大多数类 Unix 风格的操作系统之上，甚至被移植到了 Microsoft Windows 上的 Cygwin 系统中，以实现 Windows 的 POSIX 虚拟接口。
倒序打印第二行前 100 个大写字母
cat filename | head -n 2 | tail -n 1 | grep '[[:upper:]]' -o | tr -d '\n'| cut -c 1-100 | rev
在这里写下第一个爬虫程序：
23.显示维基百科主页的 html 源码.py
```
import urllib2
c = urllib2.urlopen('https://en.wikipedia.org/wiki/Main_Page')
contents = c.read()
print contents
```
64.生成[0,n)的所有子集.C
```
#include<stdio.h>
int a[];
void dfs(int n,int cur){
    for(int i=0;i<cur;++i)
        printf("%d",a[i]);
    printf("\n");
    int MIN=(cur?a[cur-1]+1:0);
    for(int i=MIN;i<n;i++){
        a[cur]=i;
        dfs(n,cur+1);
    }
}
int main(void){
```

```
    dfs(10,0);
    return 0;
}
```

17.最大连续子序列和.Cpp

现在尝试用动态规划的方法解决这个问题，设 dp[i]表示以序列中元素 A[i]作为末尾的连续子序列的最大和，那么我们最终要的就是 dp[0]到 dp[n-1]中最大的那一个，如何求解 dp 数组？首先设定边界 dp[0]=A[0]，然后设计一个状态转移方程，把 dp[i]按子问题的解来构造。如果 dp[i]中只有一个元素即 A[i]，那么 dp[i]=A[i]；如果 dp[i]中不只有一个元素，那么 dp[i]=dp[i-1]+A[i](因为已经设定了 A[i]必须作为子序列的末尾)。

```
#include<cstdio>
#include<algorithm>
using namespace std;
const int maxn=10010;
int A[maxn],dp[maxn];
int main()
{
    int n;
    scanf("%d",&n);
    for(int i=0;i<n;i++)
        scanf("%d",&A[i]);
    dp[0]=A[0];                                  //边界
    for(int i=1;i<n;i++)
        dp[i]=max(A[i],dp[i-1]+A[i]);         //状态转移方程
    int k=0;
    for(int i=1;i<n;i++)
        if(dp[i]>dp[k])         //找到最大的那个 dp[i]
            k=i;
    printf("%d\n",dp[k]);
    return 0;
}
```

算法的时间复杂度为 O(N)，要注意输入变量是运行时变量。

动态规划寻找最长回文子串的思路如下：设 dp[i][j]表示 S[i]到 S[j]的子串是不是回文，是则为 1，不是为 0，那么情况可以分为两类，如果 S[i]==S[j]，那么只要 S[i+1]到 S[j-1]是回文，S[i]到 S[j]就是回文，同样，如果 S[i+1]到 S[j-1]不是回文，S[i]到 S[j]也就不是回文；如果 S[i]!=S[j]，那么一定不是回文。

```
def manacher(s):
    s = '#'+'#'.join(s)+'#'
    RL = [0]*len(s)
    MaxRight=0
    pos=0
    MaxLen=0
    for i in range(len(s)):
        if i<MaxRight:
            RL[i] = min(RL[2*pos-i], MaxRight-i)
        else:
            RL[i]=1
        while i-RL[i]>0 and i+RL[i]<len(s) and s[i-RL[i]]==s[i+RL[i]]:
            RL[i] += 1                        #尝试扩展
        if RL[i]+i-1>MaxRight:
            MaxRight=RL[i]+i-1                #更新 MaxRight 和 pos
            pos=i
        MaxLen=max(MaxLen, RL[i])
    return MaxLen-1
```

求最长回文子串长度的最理想算法是 manacher 算法，时间复杂度为 O(N)
首先给字符串首尾和中间添加特殊字符(比如‘#’)，这样就解决了字符串长度的奇偶性所带来的对称轴位置问题。为了解决重复访问的问题，定义 RL[i]为以第 i 个字符为对称轴的回文串的半径，那么 RL[i]-1 的值正是原本那个没有插入特殊字符的字符串中以 i 为对称轴的回文串的长度。所以问题转化为求 RL 数组，基本思路是利用回文串的对称性，扩展回文串。MaxRight 表示当前访问到的所有回文子串所触及到的最右边的位置，它所对应的回文串的对称轴位置记为 pos。用一个简单的循环遍历字符串，求每个 i 位置的 RL[i]。

18.01 背包问题.Cpp

有 n 件物品，每件物品的重量为 w[i]，价值为 c[i]，现有容量为 V 的背包，问如何选取物品放入背包，使得背包内物品的总价值最大
令 dp[i][v]表示前 i 件物品($1 \le i \le n, 0 \le v \le V$)恰好装入容量为 v 的背包中所能获得的最大价值，针对第 i 件物品，有两种情况：
不放第 i 件物品进来，那就是 dp[i-1][v]
放第 i 件物品进来，那就是 dp[i-1][v-w[i]]+c[i]
由于只有这两种情况，所以 dp[i][v]就是以上两者之中取最大值
边界是 dp[0][v]=0($0 \le v \le V$)，即对于任何容量为 v 的背包，如果不放物品，那么总价值为 0。
事实上空间复杂度还能继续降低，可以把 dp[i][v]中的 i 去掉(去掉一个数组维度)，把 i 做成一个从 1 到 n 的循环，因为 dp[i][v]只和之前的状态 dp[i-1][]有关，而 w[i]和 c[i]是已知的

```
#include<cstdio>
#include<algorithm>
using namespace std;
const int maxn=100;         //物品最大件数
const int maxv=1000;                //V 的上限
int w[maxn],c[maxn],dp[maxv];
int main()
{
    int n,V;
    scanf("%d%d",&n,&V);
    for(int i=0;i<n;i++)
        scanf("%d",&w[i]);
    for(int i=0;i<n;i++)
        scanf("%d",&c[i]);
    for(int v=0;v<=V;v++)
        dp[v]=0;                //边界
    for(int i=1;i<=n;i++)
        for(int v=V;v>=w[i];v--)
            dp[v]=max(dp[v],dp[v-w[i]]+c[i]);   //状态转移方程
    int res=0;
    for(int v=0;v<=V;v++)   //寻找 dp[0...V]中的最大者就是答案
        if(dp[v]>res)
            res=dp[v];
    printf("%d\n",res);
    return 0;
}
5 8                     //输入数据
3 5 1 2 2
4 5 2 1 3
10                      //输出结果
```

两个 Java 示例程序：

```
package practice;

import java.util.*;
interface Nose{public int iMethod();}
                      //接口的定义，表示返回值是整数，接口的方法一定是抽象的
                                    //抽象类的作用就只是被继承，接口的实现
abstract class Picasso implements Nose{public int iMethod(){return 7;}}
class Clowns extends Picasso{
           //静态方法：不依靠实例变量也就不需要对象的默认做的动作
//类中包含的静态方法只要类存在就会首先执行，如果该类被继承，则父类和子类都会先后执行这个方法
    static{System.out.println("super static block");}
//构造函数，新建这个类的实例时会执行，如果有继承则也是父到子的先后顺序执行
    Clowns(){System.out.println("super constructor");}
}
class Acts extends Picasso{public int iMethod(){return 5;}}
                                    //继承的同时重写了父类的方法
class MyEx extends Exception{}
public class god extends Clowns{
    public static void main(String[] args){
        Random random;
        random=new Random();         //这个不是 Math 类中的 Random
        for(int x=0;x<30;x++)
            System.out.print(random.nextInt(8));
        System.out.print("\n");
        int[] a={1,3,4,6,7,8,12,14,17,23};
        System.out.println(Arrays.binarySearch(a,3,4,8));
                         //在 a[3]到 a[4]中查找 8，找不到就返回-(插入点+1)
        float d=3.5f;         //后面不加 f 就会导致默认的双精度截断为单精度
        Nose[] i=new Nose[3];
        i[0]=new Acts();
        i[1]=new Clowns();    //新建了一个父类的实例：父类的构造函数执行一次
        i[2]=new god();
    //新建了一个子类的实例，父类的构造函数执行一次，子类的构造函数执行一次
        for(int x=0;x<3;x++)
            System.out.println(i[x].iMethod()+" "+i[x].getClass());
        final int DAYIM=1000*60*60*24;                 //一天的毫秒数
        Calendar c=Calendar.getInstance();
        c.set(2004,0,7,15,40);
        long day1=c.getTimeInMillis();
        for(int x=0;x<3;x++){
            day1+=(DAYIM*29.52);         //以 29.52 的周期计算满月
            c.setTimeInMillis(day1);
            System.out.println(String.format("full moon on %tc", c));
        }                                              //格式化输出
        System.out.println(String.format("%,.2f", 465734562.0978));
        String test="yes";               //如果不是"yes"，输出 throws
        try{
            System.out.print("t");
            doRisky(test);
            System.out.print("o");
        }catch(MyEx e){
            System.out.print("a");
```

```
        }finally{
            System.out.print("w");
        }
        System.out.println("s");
        System.out.println("in main");
        god st=new god();
    }
    static void doRisky(String t) throws MyEx{
        System.out.print("h");
        if("yes".equals(t))
            throw new MyEx();
        System.out.print("r");
    }
    static int rand;
    static{                        //继承的同时重写的静态方法和构造函数
        rand=(int)(Math.random()*6);
        System.out.println("static block "+rand);
    }
    god(){System.out.println("constructor");}
}
```

输出：

```
super static block
static block 3
347002730523311646517112721314
-5
super constructor
super constructor
constructor
5 class practice.Acts
7 class practice.Clowns
7 class practice.god
full moon on 星期五 二月 06 04:09:18 CST 2004
full moon on 星期六 三月 06 16:38:06 CST 2004
full moon on 星期一 四月 05 05:06:54 CST 2004
465,734,562.10
thaws
in main
super constructor
constructor
```

```
class BankAccount{
    private int balance=100;     //账户一开始有 100 元
    public int getBalance(){return balance;}
    public void withdraw(int amount){balance-=amount;}
}
public class god implements Runnable{
    private BankAccount account=new BankAccount(); //只有一个共享账户
    public static void main(String[] args){
        god theJob=new god();        //任务初始化
        Thread one=new Thread(theJob);   //使用相同任务的两个线程
        Thread two=new Thread(theJob);   //两个线程存取同一个账户
        one.setName("Alice");     //启动并命名线程
        two.setName("Bob");
        one.start();
        two.start();
```

```
    }
    public void run(){        //观察两个线程执行 run
        for(int x=0;x<10;x++){    //Runnable 接口中的 run()类似于 main()方法
            makeWithdrawal(10);
            if(account.getBalance()<0)
                System.out.println("Overdrawn!");
        }
    }
    public String name(){return Thread.currentThread().getName();}
//使用 synchronized 这个关键字来修饰方法使它每一次只能被单一的线程使用
    private synchronized void makeWithdrawal(int amount){
        if(account.getBalance()>=amount){
//检查账户余额，如果透支就列出信息，不然就去睡一会儿，然后醒来完成提款操作
            System.out.println(name()+" is about to withdraw");
            try{
                System.out.println(name()+" is going to sleep");
                Thread.sleep(10);
            }catch(InterruptedException e){
                e.printStackTrace();
            }
            System.out.println(name()+" woke up");
            account.withdraw(amount);
            System.out.println(name()+" completes the withdraw");
        }else{
            System.out.println("Sorry,not enough for "+name());
        }
    }
}
```

输出：

```
Alice is about to
withdraw
Alice is going to
sleep
Alice woke up
Alice completes the
withdraw
Alice is about to
withdraw
Alice is going to
sleep
Alice woke up
Alice completes the
withdraw
Alice is about to
withdraw
Alice is going to
sleep
Alice woke up
Alice completes the
withdraw
Bob is about to
withdraw
Bob is going to sleep
Bob woke up
Bob completes the
withdraw
Bob is about to
withdraw
Bob is going to sleep
Bob woke up
Bob completes the
withdraw
Bob is about to
withdraw
Bob is going to sleep
Bob woke up
Bob completes the
withdraw
Bob is about to
withdraw
Bob is going to sleep
Bob woke up
Bob completes the
withdraw
Bob is about to
withdraw
Bob is going to sleep
Bob woke up
Bob completes the
withdraw
Bob is about to
withdraw
Bob is going to sleep
Bob woke up
Bob completes the
withdraw
Sorry,not enough for
Bob
Sorry,not enough for
Bob
Sorry,not enough for
Bob
Sorry,not enough for
Alice
Sorry,not enough for
Alice
Sorry,not enough for
Alice
Sorry,not enough for
Alice
Sorry,not enough for
Alice
Sorry,not enough for
Alice
Sorry,not enough for
Alice
```

Python 中列表是一个可迭代对象，用的是方括号；生成器是一个只能读取一次的可迭代的实时生成的数据，用圆括号；yield 是一个类似于 return 的关键字，只是这个函数返回一个生成器，当你调用这个函数时，函数内部的代码并不执行，当你用 for 迭代时才会执行

Python 中的闭包使局部变量在函数外被访问（调用），闭包的后期绑定会导致 late binding

```
def testfun():
    return [lambda x : i*x for i in range(4)]        #6 6 6 6
    #return (lambda x : i*x for i in range(4))       #0 2 4 6
    #return [lambda x, i=i : i*x for i in range(4)]   #0 2 4 6
    #for i in range(4):yield lambda x: i*x           #0 2 4 6
for everylambda in testfun():
    print everylambda(2),
```

numpy 用法集：

```
import numpy as np
print(np.__version__)          #打印 numpy 的版本号
a = np.array([1,2,3,4,9,5])
b = np.array([4,5,6,7,9,8])
print(np.setdiff1d(a,b))       #打印 a 中不属于 b 的元素
print(np.where(a==b))          #打印 a 和 b 相同元素的相同位置
ex = np.arange(9).reshape(3,3)
print(ex)
print(ex[:,[1,0,2]])           #交换第 1,2 列
print(ex[[1,0,2],:])           #交换第 1,2 行
print(ex[::-1])                #行反转
print(ex[:,::-1])              #列反转
c=np.arange(20)
np.set_printoptions(threshold=6)    #打印时只显示 6 个元素，中间的省略
print(c)
d=np.array([[1,np.nan,3],[np.nan,5,6],[7,8,np.nan]])
print(np.isnan(d).any())           #矩阵中是否有缺失值
print(np.isnan(d).sum())           #缺失值的数目
print(np.where(np.isnan(d)))       #缺失值的位置
print(np.clip(c,a_min=7,a_max=15))  #小于 7 的数改成 7，大于 15 的数改成 15
print(d[~np.isnan(d)].reshape(3,-1))    #去掉缺失值，重新设定矩阵规格
```

```
#include<bits/stdc++.h>
using namespace std;
int main(void){
    int n=5,r=2;
    for(int s=(1<<r)-1;s<1<<n;){
        cout<<(bitset<5>)s<<endl;
        int x=s&-s;
        int y=s+x;
        s=((s&~y)/x>>1)|y;
    }                       //枚举大小为 r 的子集
    cout<<"gap"<<endl;
    int d=15;               //枚举 d 的子集
    for(int i=d;i;i=(i-1)&d)
        cout<<(bitset<4>)i<<endl;
}
```

```
00011
00101
00110
01001
01010
01100
10001
10010
10100
11000
gap
1111
1110
1101
1100
1011
1010
1001
1000
0111
0110
0101
0100
0011
0010
0001
```

面向对象中两种基本的设计模式：

单例设计模式：确保一个类只有一个实例，并且只能通过类的内部方法访问此实例

工厂方法设计模式：工厂方法提供接口以创建某个类的实例，由子类决定实例化哪个类。实现时，你可以将创建器(Creator)设计为抽象类型，不为工厂方法提供具体实现；或者，创

建器类是实体类，为工厂方法提供具体实现。在这种情况下，工厂方法需要传入参数，代表该实例化哪个类
XML(可扩展标记语言)用于简化 Internet 的文档信息传输，大多数编程语言都可以解析它
更多的 Linux 命令：
ps 查看系统进程，有一列是 STAT，为 S 表示正在休眠，s 表示主进程，Z 表示僵尸进程
ps －aux 或 ps －elf 查看当前有哪些进程
top 查看系统资源占用情况
vmstat 查看进程状态以及内存、磁盘、交换区正在交换的数据大小
简述 raid0 raid1 raid5 三种工作模式的工作原理及特点
RAID,可以把硬盘整合成一个大磁盘,还可以在大磁盘上再分区,放数据。还有一个大功能，多块盘放在一起可以有冗余（备份）RAID 整合方式有很多，常用的：0 1 5 10
RAID 0，可以是一块盘和 N 个盘组合 其优点读写快，是 RAID 中最好的。缺点：没有冗余，一块坏了数据就全没有了
RAID 1，只能 2 块盘，盘的大小可以不一样，以小的为准 10G+10G 只有 10G，另一个做备份。它有 100%的冗余，缺点：浪费资源，成本高
RAID 5 ，3 块盘，容量计算 10*（n-1）,损失一块盘特点，读写性能一般，读还好一点，写不好
冗余从好到坏：RAID1 RAID10 RAID 5 RAID0 性能从好到坏：RAID0 RAID10 RAID5 RAID1
成本从低到高：RAID0 RAID5 RAID1 RAID10
单台服务器：很重要盘不多，系统盘，RAID1
数据库服务器：主库：RAID10 从库 RAID5\RAID0（为了维护成本，RAID10）
WEB 服务器，如果没有太多的数据的话，RAID5,RAID0（单盘）有多台，监控、应用服务器，RAID0 RAID5 我们会根据数据的存储和访问的需求，去匹配对应的 RAID 级别

```
#include<bits/stdc++.h>       //C/C++输出 1 到 100 的 3 种方法
#define M printf("%d\n",i++);
#define K M M M M M M M M M M
#define N K K K K K K K K K K    //宏展开
static int count=1;
class T{                         //数组构造函数
    public: T(){std::cout<<count++<<std::endl;}
};
template <unsigned n> void print(){
    print<n-1>();                //模板元编程
    std::cout<<n<<std::endl;
}
template<> void print<1>(){std::cout<<1<<std::endl;}
int main(){
    int i=1;N;
    T *t=new T[100];
    print<100>();
    return 0;
}
```

把一个整数的二进制表示中的奇数位置和偶数位置的 bit 交换：

```
int swapOddEvenBits(int x){
     return (((x&0xAAAAAAAA)>>1)|((x&0x55555555)<<1));
}
```

计算机网络从组成来看，分为硬件、软件、协议。从功能来看，分为资源子网、通信子网。从传输方式来看，分为广播式网络(所有连网计算机共享一个公共信道)、交换式网络(如果两个计算机没有直接连接，就需要选择中间结点来存储并转发，这个选择机制叫做路由)。从覆盖的地理范围来看，分为广域网(采用交换技术)、局域网(采用广播技术)。

TCP/IP 模型	ISO/OSI 参考模型	PDU	SAP	协议	主要功能	设备
应用层	应用层		用户界面	FTP SMTP POP3 HTTP DHCP RIP BGP		
	表示层				数据格式转换	
	会话层				会话管理、同步	
传输层 (无连接和面向连接)	传输层 (面向连接)	报文	端口号(套接字 socket)	TCP UDP	两个用户进程之间端到端通信	
网际层 (无连接)	网络层 (无连接和面向连接)	分组	IP 地址	IP(两个版本：Ipv4/Ipv6) ICMP ARP OSPF IGMP	根据路由算法选择路径 差错控制、流量控制、拥塞控制(过量的分组引起网络性能下降)	路由器
网络接口层	链路层	帧	MAC 地址(网卡地址)	PPP HDLC	两个主机之间点到点通信 组帧、差错控制、流量控制(使发送速度等于接收速度)	网桥、交换机
	物理层	比特	网卡接口		在媒体设备上直接传输比特来通信	中继器、集线器、调制解调器

客户机是客户访问网络的出入口，服务器是提供服务、存储信息的设备。网络中必然存在充当这两个角色的计算机，一个网络中可以没有数据库系统

目前的国际通用网络模型是分层的，相邻的两层通过服务访问点(SAP)进行交互，越是底层数字越少，N 层的 SAP 就是 N+1 层接收 N 层提供的服务的地方，最底层只提供服务，最高层向用户提供服务。每一层传输的数据单位称为该层的协议数据单元(PDU)

网络提供的服务可以分为面向连接服务(通信之前双方必须确认建立连接，通信之后双方必须确认释放连接)、无连接服务(尽最大努力交付，即需要发送时就直接发送)。也可以分为可靠服务(具有纠错检错机制，保证数据正确)、不可靠服务

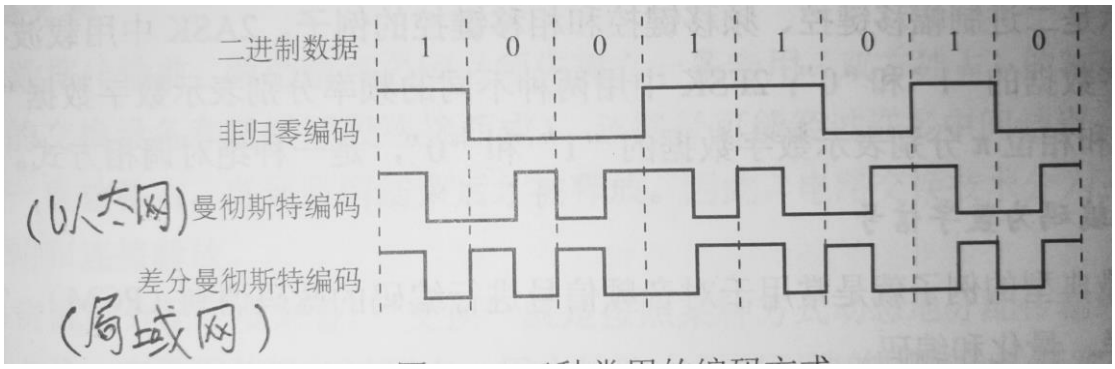
通过自我复制实现的恶意软件：病毒(需要用户做点什么，哪怕用户完全不知情)、蠕虫(不需要用户交互)

物理层

码元：数字信号的基本单位，波特率：单位时间内传输的码元数目　比特率：单位时间内传输的比特数目。数据率可以用这两个指标来表示

N 种不同的码元→码元的离散电平数目为 N→需要$\log_2 N$个二进制位→一个码元携带$\log_2 N$个 bit 的信息量→比特率=波特率×$\log_2 N$

数字信号 ←(编码)← 数据 →(调制)→ 模拟信号



交换技术分为 3 种：

	电路	报文	分组
连接延迟	有	无	无
转发延迟	无	有，较大	有，并行性，较小
传输延迟	有，很小	有，较大	有，较小

信噪比=$10\log_{10}(S/N)$　S:信道传输信号的平均功率　N:信道内部的噪声功率

奈奎斯特定理：最大波特率=$\mathrm{W}\times 2$

香农定理：最大比特率=$\mathrm{W}\times\log_2(1+\frac{S}{N})$　　W：信道的带宽

分组交换分为 2 种：

	数据报	虚电路
是否要建立连接	否	是
目的地址	每个分组都有	一开始有，然后用虚电路号
路由	每个分组独立	同一条虚电路用同一个路由
分组有序到达	不保证	保证
可靠性由谁保证	用户自己	网络
故障适应	出故障的丢掉分组，其他分组改变路径，继续	所有经过故障点的虚电路都不能再正常工作

中继器的作用是把衰减的信号再生并转发出去

集线器实际上是一个多端口的中继器，从一个端口收到数据后，从其他所有端口转发出去

链路层

为了避免丢帧，采用计时器超时重发的方式

采用滑动窗口来进行流量控制，把要发送和要接收的帧编号排成一排，就可以想象成是发送窗口和接收窗口在这一排上滑动

	发送窗口大小	接收窗口大小
停止-等待协议	1	1
后退 N 帧协议	>1	1
选择重传协议	>1	>1

当接收窗口大小为 1 时，可以保证帧的有序接收

后退 N 帧协议中，只要发送方收到了第 n 帧的确认，意思就是第 n 帧之前包括第 n 帧的所有帧都已经收到(累积确认)。选择重传协议中，选择那些超时的帧重新发送

介质访问控制：采取一定措施，使两对结点之间的通信不会互相干扰

码分多址(CDMA)靠不同的编码来区分各路原始信号

CSMA/CA：发送数据时先广播告知其他结点，让其他结点在某段时间内不要发送数据，以免碰撞。用于无线局域网

CSMA/CD：发送之前就听，边发边听，听到碰撞了马上停止发送。用于总线式以太网，只能进行半双工通信

假设 t 是信号的单程传播延迟，那么在发送帧之后最多 2t 时间内就能知道发送的帧是否遭到碰撞。为了确保发送帧的同时能检测到碰撞，必须满足帧的传输延迟≥ 2t，

以太网就是一个很大的局域网，相距较远的局域网通过路由器和广域网连接

广域网的两种链路层控制协议：PPP(面向字节)、HDLC(面向比特)

两个以太网通过网桥连接成更大的以太网，交换机相当于一个多端口的网桥

可能形成冲突的网卡站点的集合叫做冲突域，能够收到一个网卡发出的信息的所有网卡的集合叫做广播域

<table>
<tr><th></th><th>能否隔离冲突域</th><th>能否隔离广播域</th></tr>
<tr><td>集线器</td><td rowspan="2">不能</td><td rowspan="4">不能</td></tr>
<tr><td>中继器</td></tr>
<tr><td>交换机</td><td rowspan="3">能</td></tr>
<tr><td>网桥</td></tr>
<tr><td>路由器</td><td>能</td></tr>
</table>

对于普通 10Mb/s 的共享式以太网，如果一共有 N 个用户，则每个用户占有的平均带宽只有总带宽的 1/N。但使用交换机时，用户在通信时是独占而不是和其他用户共享带宽，因此每个用户仍然可以得到 10Mb/s 的带宽，这是交换机的最大优点

36．主机甲与主机乙之间使用后退 N 帧协议(GBN)传输数据，甲的发送窗口尺寸为 1000，数据帧长为 1000 字节，信道带宽为 100Mbps，乙每收到一个数据帧立即利用一个短帧(忽略其传输延迟)进行确认，若甲、乙之间的单向传播延迟是 50ms，则甲可以达到的最大平均数据传输速率约为_____。

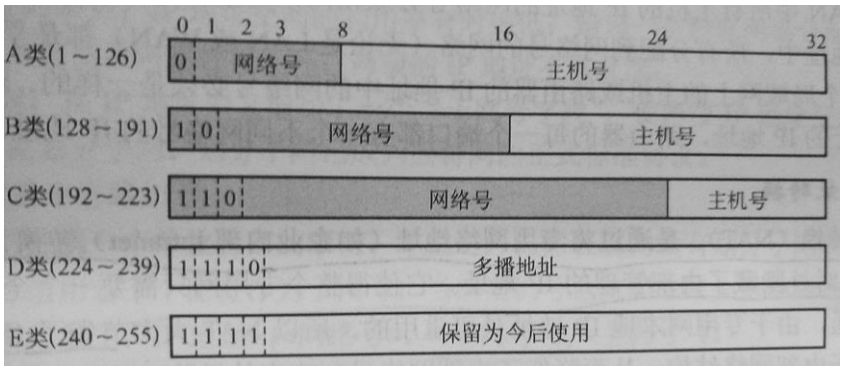
A．10Mbps　　B．20Mbps　　C．80Mbps　　D．100Mbps

因为是后退 N 帧协议，所以要考虑发送一个数据到收到它的确认的这段时间内最多能发送多少数据，这段时间是单向传播延迟的两倍，而一次最多能发送一个窗口大小，并且答案最大不会超过信道带宽，所以是$(1000\times1000\times8)\text{bit}\div(50\times2)\text{ms}=80\times10^6 bit/s$　　选 C

网络层

慢收敛是形成路由回路的根本原因

IP 数据报中首部长度、总长度、片偏移的基本单位分别是 4B、1B、8B



主机号全 0 表示本网络

主机号全 1 表示本网络的广播地址

IP 地址&子网掩码=子网的网络地址

网络类别	最大可用网络数	第一个可用的网络号	最后一个可用的网络号	每个网络中的最大主机数
A	2^7-2	1	126	2^{24}-2
B	2^{14}-1	128.1	191.255	2^{16}-2
C	2^{21}-1	192.0.1	223.255.255	2^8-2

最初设计的分类 IP 地址，由于每类地址所能连接的主机数大大超过一般单位的需求量，从而造成了 IP 地址的浪费。划分子网通过从网络的主机号借用若干比特作为子网号，从而对原来较大规模的网络细分成几个规模较小的网络，提高了 IP 地址的利用率。

CIDR 是比划分子网更为灵活的一种手段，它消除了 A、B、C 类地址以及划分子网的概念。使用各种长度的网络前缀来代替分类地址中的网络号和子网号，将网络前缀都相同的 IP 地址组成“CIDR 地址块”。网络前缀越短，地址块越大。因特网服务提供者再根据客户的具体情况，分配合适大小的 CIDR 地址块，从而更加有效地利用了 IPv4 的地址空间。

采用网络地址转换 NAT，可以使一些使用本地地址的专用网连接到因特网上，从而使得一些机构的内部主机可以使用专用地址，只需给此机构分配一个 IP 地址即可，并且这些专用地址是可重用的——其他机构也可使用，所以大大节省了 IP 地址的消耗。

尽管以上三种方法可以在一定阶段内有效缓解 IP 地址耗尽的危机，但无论从计算机本身发展还是因特网规模和传输速率来看，现在的 IPv4 地址已很不适用了，所以治本的方法还是使用 128bit 编址的 IPv6 地址。

ARP（地址解析协议）：完成 IP 地址到 MAC 地址的映射

DHCP（动态主机配置协议）：提供即插即用的机制，允许新的计算机加入网络并动态获址

ICMP（网际控制报文协议）：让主机和路由器能够报告错误

3 种路由协议：

	RIP	OSPF	BGP
类型	内部		外部
路由算法	距离-向量	链路状态	路径-向量
传递协议	UDP	IP	TCP
路径选择	跳数最少(≤ 15)		较好，非最佳
交换结点	相邻的路由器	网络中所有路由器	相邻的路由器
交换内容	本路由器知道的全部信息，即自己的路由表	相邻的所有路由器的链路状态	第一次是整个路由表，然后是发生了变化的部分

传输层

UDP 是一个简单的协议，它提供无连接的不可靠服务，仅仅在 IP 协议上增加了复用和分用、差错检测。远程登录不适合使用 UDP 协议

TCP 是面向连接的协议，TCP 连接的建立采用客户/服务器模式

SYN=1：连接请求　　FIN=1：释放请求　　ACK=1：确认已连接

seq：起始序号（可看做随机的）　　ack：确认序号

对上述 TCP 连接建立和释放的总结如下：

（1）连接建立

分为 3 步：

① SYN=1，seq=x。

② SYN=1，ACK=1，seq=y，ack=x+1。

③ ACK=1，seq=x+1，ack=y+1。

（2）释放连接

分为 4 步：

① FIN=1，seq=u。

② ACK=1，seq=v，ack=u+1。

③ FIN=1，ACK=1，seq=w，ack=u+1。

④ ACK=1，seq=u+1，ack=w+1。

超时和冗余 ACK 会导致 TCP 对报文段进行重传

发送方根据它对当前网络的拥塞状况的估计而确定的窗口大小就是拥塞窗口，接收方根据自己的缓存状况动态的调整发送方的窗口大小就是接收窗口

发送方的发送窗口的实际大小是

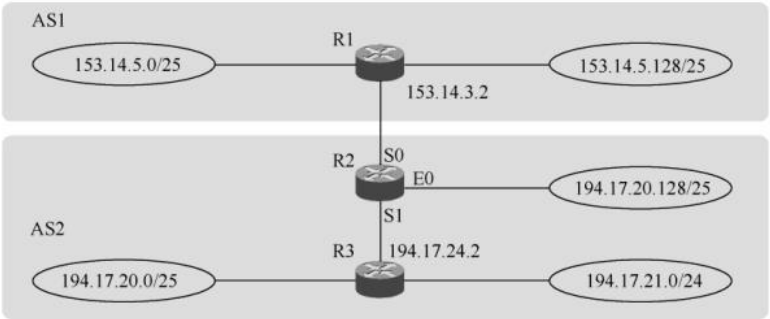
min(拥塞窗口，接收窗口)

面向字节的意思是对每个字节编号，并不是对每个接收到的字节都发回确认，而是发送一个完整的报文段后才发回一个确认。也就是说，TCP 是面向字节的，采用对报文段的确认机制

假如 A 和 B 已经建立 TCP 连接，A 向 B 发送报文中包含 seq 和 ack，那么 B 发送回来的是

seq = ack　　ack = seq +数据部分字节数

47.（9 分）假设 Internet 的两个自治系统构成的网络如题 47 图所示，自治系统 ASI 由路由器 R1 连接两个子网构成；自治系统 AS2 由路由器 R2、R3 互联并连接 3 个子网构成。各子网地址、R2 的接口名、R1 与 R3 的部分接口 IP 地址如题 47 图所示。



题 47 图　网络拓扑结构

请回答下列问题。

（1）假设路由表结构如下表所示。请利用路由聚合技术，给出 R2 的路由表，要求包括到达题 47 图中所有子网的路由，且路由表中的路由项尽可能少。

目的网络	下一跳	接口

（2）若 R2 收到一个目的 IP 地址为 194.17.20.200 的 IP 分组，R2 会通过哪个接口转发该 IP 分组？

（3）R1 与 R2 之间利用哪个路由协议交换路由信息？该路由协议的报文被封装到哪个协议的分组中进行传输？

（1）在 AS1 中，子网 153.14.5.0/25 和子网 153.14.5.128/25 可以聚合为子网 153.14.5.0/24；在 AS2 中，子网 194.17.20.0/25 和子网 194.17.21.0/24 可以聚合为子网 194.17.20.0/23，但缺少 194.17.20.128/25；子网 194.17.20.128/25 单独连接到 R2 的接口 E0。

于是可以得到 R2 的路由表如下：

目的网络	下一跳	接口
153.14.5.0/24	153.14.3.2	S0
194.17.20.0/23	194.17.24.2	S1
194.17.20.128/25	—	E0

（2）该 IP 分组的目的 IP 地址 194.17.20.200 与路由表中 194.17.20.0/23 和 194.17.20.128/25 两个路由表项均匹配，根据最长匹配原则，R2 将通过 E0 接口转发该 IP 分组。

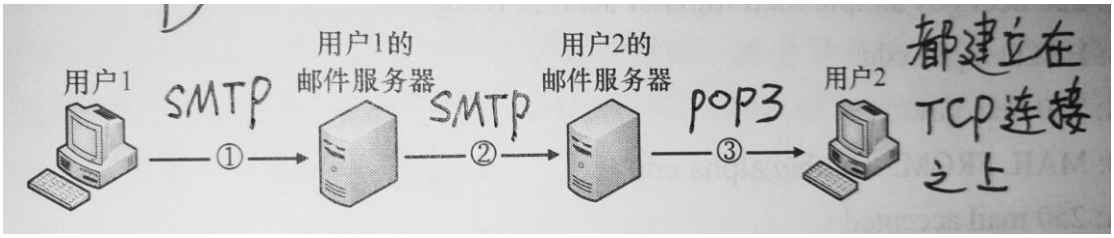
（3）R1 与 R2 之间利用 BGP4（或 BGP）交换路由信息；BGP4 的报文被封装到 TCP 协议段中进行传输。

应用层

DNS 系统采用客户/服务器模型，协议运行在 UDP 之上，使用 53 号端口，最后一个.com 叫做顶级域名，多个域名可以指向同一主机 IP 地址，同一域名在不同时间可能解析出不同的 IP 地址。这个解析由运行在域名服务器的程序完成

FTP（文件传输协议）建立在 TCP 连接之上，允许客户指明文件的类型和格式

发送电子邮件时用到的协议：



万维网（WWW）是一个资料空间，在这个空间中，有用的对象叫做资源，并且有一个全空间的统一资源定位符（URL）来标记，这些资源通过超文本传输协议（HTTP）传送给使用者，使用者通过点击链接获取资源

一个包含 100 个图片的网站页面，服务器要传输 1 个基本 HTML 文件和 100 个图片，非持续连接（HTTP 请求报文中是 `Connection:Close`，反之为 `Keep-alive`）时需要打开和关闭 TCP 连接 101 次

使用 HEAD 方法时服务器可以对 HTTP 报文进行响应，主要作用是调试。请求报文中有 cookie 报头表示曾经访问过。

2.5亿个整数中找出只出现一次的整数的个数，注意内存空间不足以容纳2.5亿个整数，且这些整数都是32位整数

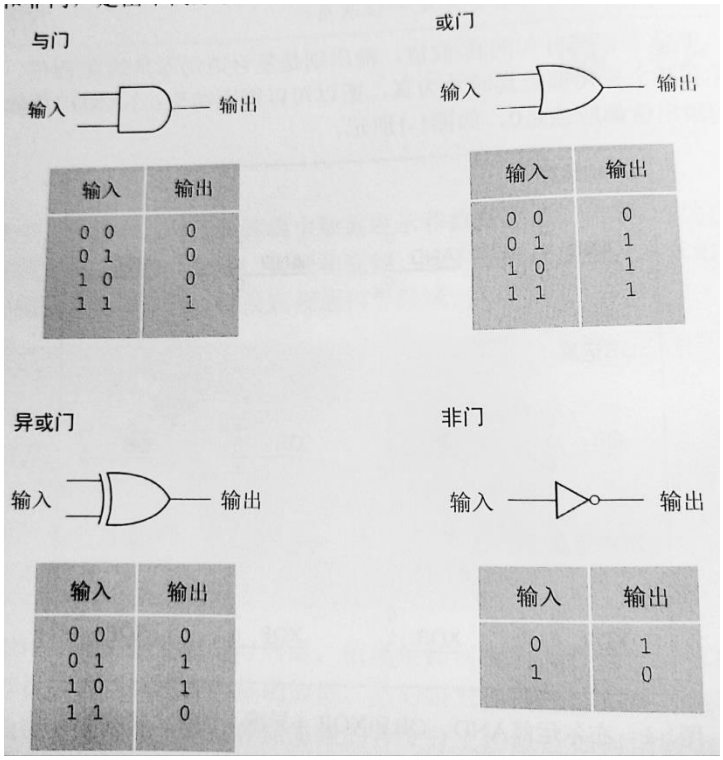
解题思路

使用Bitmap，所有整数共月44亿不到，44亿个比特位大约需要522M空间，我们使用2个Bitmap表示1个数，这样大约需要1G空间。这2个Bitmap相同下标下的位构成几种情况，00表示未出现，01表示出现1次，11表示出现2次及以上。遍历这些数，如果对应的位置是00，则将其置01；如果是01，将其置11；如果是11，保持不变。 对应关系函数是：

```
int position = 0;//bitmap的下标
if(整数 < 0){
    position = 整数 * -1;
}else if(整数 > 0){
    position = 整数 + Integer.MAX_VALUE + 1;
}else{
    position = 0;
}
```

术语抽象在 CS 中是指一个实体的外部特征与其内部构成的细节之间的分离

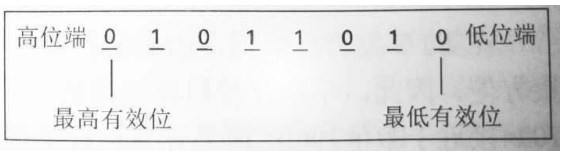
门指的是一种设备，给定一种布尔运算的输入值，门可以得出该运算的输出值。现代计算机中，门是由小型电子电路实现的



触发器是存储器的基本部件，它是一个可以产生 0 或 1 输出值的电路，它的值会一直保持不变，直到有另一个电路过来的临时脉冲(临时变成 1 之后再变为 0)使其变换成其他值。换句话说，通过设置可以让输出在外界刺激的控制下"记住 0 或者 1"

超大规模集成技术将数百万个电子元件构造在芯片上，从而创建出包含数百万个触发器及其控制电路的微型设备

二进制数字组成的串也称为位模式，虽然计算机中没有左右的概念，但是我们通常假设存储单元的位是排成一行的



该行的左边称为高位端，右边称为低位端

因为主存中的存储单元像数组元素一样可以直接寻址，主存储器也被称为随机存取存储器(**RAM**)，除了主存以外，我们还可借助海量存储器，它通常有 3 种技术类型：磁技术(旋转磁盘，移动读/写磁头。磁盘的上的多个同心圆称为磁道，一个磁道可以划分成多个扇区)、光技术(激光扫描检测反射层的不规则反射偏差，用于数字音频光盘 CD、数字多功能光碟 DVD。蓝光技术使用更精确的蓝色激光)、闪存技术(与前两者相比，不需要物理运动来存取，二进制数字直接由电子信号发送到存储介质中，介质中二氧化硅的微小晶格截获了电子，从而转换了微电子电路的性质。用于固态硬盘 SSD、安全数字存储卡 SD 卡)

一个存放在存储器中的对象，如果取它的地址，只会取到最前面的那个地址，而存放一个对象往往需要多个存储单元(地址)。最低有效字节在最前面的方式，称为小端法，反之亦然

在 C 语言中，以 0x 或 0X 开头的数字被看做是十六进制数

十六进制数字	0	1	2	3	4	5	6	7
十进制值	0	1	2	3	4	5	6	7
二进制值	0000	0001	0010	0011	0100	0101	0110	0111
十六进制数字	8	9	A	B	C	D	E	F
十进制值	8	9	10	11	12	13	14	15
二进制值	1000	1001	1010	1011	1100	1101	1110	1111

十六进制→二进制：把每一位十六进制数字转成二进制位，按原来的顺序拼起来

二进制→十六进制：从最右边开始往左，按每 4 位一组划分，如果最左边不够 4 位就用 0 补上，每一组转成对应的十六进制数字

十进制→十六进制：把十进制的 x 做$x = q \times 16 + r$，然后取十六进制表示的 r 作为最低位，再反复把 q 赋值给 x 重复以上过程

十六进制→十进制：共 n 位，最右边为第 0 位，结果是$\sum_{k=0}^{n}$(*第k位上的数对应的十进制数*$\times 16^k$)

特殊情况：当十进制数 x 是 2 的非负整数 n 次幂时

→二进制：1 后面跟 n 个 0

→十六进制：$n = i + 4j$，其中$0 \le i \le 3$，转换后的数可以写成 0X $+ 2^i +$ j 个 0

位向量的一个用途是表示有限集合，我们可以用位向量$[a_{w-1}, \dots, a_1, a_0]$编码任何子集$A \subseteq \{0,1,\dots,w-1\}$，其中$a_i = 1$当且仅当$i \in A$。例如 01101001 表示集合$\{0,3,5,6\}$，使用这种方法，布尔运算**|**和**&**分别对应于集合的并和交，**~**对应于集合的补。

逻辑运算认为所有非 0 参数都表示 true，而参数 0 表示 false，例如**!**0x00=0x01

Java 中的右移运算：算术右移 x>>k，在左端补 k 个最高有效位的值；逻辑右移 x>>>k，在左端补 k 个 0

十进制小数→任意进制：整数部分采用除基取余法，最先取得的余数作为最低位，商为 0 时结束；小数部分采用乘基取整法，最先取得的整数作为最高位，乘积为 0 时结束，最后把两部分拼接起来。

很明显，并非任何十进制小数都可以用二进制表示。

二进制小数→八进制或十六进制：以小数点为界，整数部分从右往左，分为 3 位一组(八进制)或 4 位一组(十六进制)，最左边可以加 0 补齐；小数部分从左往右，类似的做法。然后按每一组转换，把两部分拼接起来

校验码的原理是通过增加一些二进制位来检验或纠错，两个位模式之间不相同的位的个数称为码距，码距≥ 2的校验码具有检错能力，并且检错能力≥纠错能力。校验码=校验位(增加的)+信息位(原来待检验的)

奇偶校验位的原理是，在原来的信息位上加一个校验位，它的取值 0 或 1 会使整个校验码中 1 的个数为奇数(奇校验码)或偶数(偶校验码)，可以检测出奇数个数的位错误，但不能确定出错的位置

用一位奇偶校验位一定能检测出一位主存错误。如果被操作的每个位模式都有奇数个 1，但却遇到了有偶数个 1 的模式，那么一定是出错了。

海明纠错码的原理是，在信息位(最低位设为 1 号位，共 n 位)的 1，2，4，8，…，号位置上插入校验位(设为 k 位)，可见，$n + k + 1 \le 2^k$，如果想再发现一位错，就要增加 1 个校验位，这时满足的关系是$n + k + 1 \le 2^{k-1}$，海明码最多能发现 2 位错并纠正 1 位错

摩尔斯电码的编码规则是：划(dash)的长度等于点(dot)的长度的 3 倍，单个的点或划之间用长度与点的长度相等的空格分开，单词中的各个字母之间用长度等于划的长度的空格分开，单个单词之间用长度等于 2 倍的划的长度的空格分开。

二进制的加法规则是：$1 + 1 = 10$，加出 10 来了就要进一位

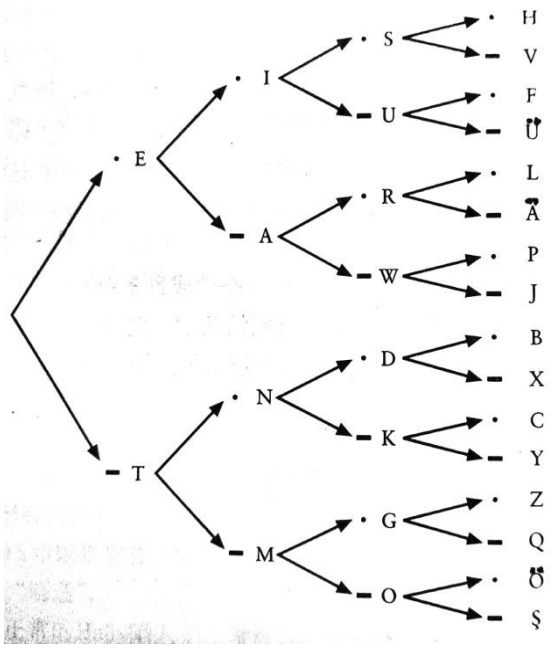
在计算机中参与运算的机器数有两类：无符号数和有符号数

无符号数的整个的字长(k 位)的全部二进制位均为数值位，没有符号位，真值的表示范围是$0 \sim 2^k - 1$；有符号数用 0 表示正，1 表示负，并约定最高位是符号位。有符号数的两种主要的机器表示法是原码和补码。真值 X 的补码表示记为$[X]_{补}$

根据是否约定小数点的位置固定不变，机器数分为定点数和浮点数。

A	·—	J	·———	S	···
B	—···	K	—·—	T	—
C	—·—·	L	·—··	U	··—
D	—··	M	——	V	···—
E	·	N	—·	W	·——
F	··—·	O	———	X	—··—
G	——·	P	·——·	Y	—·——
H	····	Q	——·—	Z	——··
I	··	R	·—·		

1	·————	6	—····
2	··———	7	——···
3	···——	8	———··
4	····—	9	————·
5	·····	0	—————



如果你想装逼，让别人看不懂你的代码，你可以把 x==y 改成!(x^y)。

定点小数：小数点在符号位和最高数值位之间；定点整数：小数点在最低数值位的右边

补码表示法：（用$\underset{X}{\rightarrow}$表示位模式，最高有效位解释为负权重）

$$\mathrm{B2}T_w\left(\underset{X}{\rightarrow}\right) = -x_{w-1}2^{w-1} + \sum_{i=0}^{w-2} x_i 2^i$$

原码表示法：（用$\underset{X}{\rightarrow}$表示位模式，最高有效位是符号位，用来确定剩下的位表示的值的正负）

$$\mathrm{B2}S_w\left(\underset{X}{\rightarrow}\right) = (-1)^{x_{w-1}} \times (\sum_{i=0}^{w-2} x_i 2^i)$$

由于在计算机中，数和地址都可以表示为十六进制，为了区分它们，数的后面加上大写的 H，地址的前面加上 0x（和 C 语言中的表示不同），十六进制表示中一位相当于二进制的 4 位。C 语言中如果要表示一个无符号数，必须在末尾加上 U 或者 u。

强制类型转换的结果保持位模式不变，只是改变了解释这些位的方式。C 语言标准规定，转换类型时，先改变大小，再完成有符号和无符号之间的改变。改变大小的方法是截断或扩展高位，保留低位。扩展无符号数只需在高位添加 0，扩展有符号数要在高位添加最高有效位（符号位），截断一个无符号数 x 到剩下 k 位的结果相当于计算$x \bmod 2^k$。有符号数转到无符号数的方法是将解释这些位的方式从补码法转到普通的二进制表示法

当执行一个运算时，如果它的一个操作数是有符号的而另一个是无符号的，那么 C 语言会隐式地将有符号数强制类型转换为无符号数

位运算的一个常见用法是实现掩码运算，这里掩码是一个位模式，表示从一个数字中选出的位的集合，例如掩码 0xFF 表示一个字的低位字节，x&0xFF 取 x 的最低有效字节，其他位全部被置为 0，表达式~0 生成一个全 1 的掩码，不管机器的字长是多少，也就是说它是可移植的 。边界对齐存储的意思是，不会把一个数据紧跟在另一个数据后面存储填满存储空间，而是为了便于寻址，把存储空间稍微空一些出来，每一个存储空间存放明确的数据，或者单纯就是强迫症，不喜欢奇数，想凑个偶数。

某计算机存储器按字节编址，采用小端方式存放数据。假定编译器规定 int 和 short 型长度分别为 32 位和 16 位，并且数据按边界对齐存储。某 C 语言程序段如下：

```
struct{
    int     a;
    char    b;
    short   c;
}record;
record.a=273;
```

若 record 变量的首地址为 0xC008，则地址 0xC008 中内容及 record.c 的地址分别为（ A ）。

A. 0x00、0xC00D　　B. 0x00、0xC00E

C. 0x11、0xC00D　　D. 0x11、0xC00E

尽管 record 大小为 7 个字节（成员 a 有 4 个字节，成员 b 有 1 个字节，成员 c 有 2 个字节），由于数据按边界对齐方式存储，故 record 共占用 8 个字节。record.a 的十六进制表示为 0x00000111，由于采用小端方式存放数据，故地址 0xC008 中内容应为低字节 0x11；record.b 只占 1 个字节，后面的一个字节留空；record.c 占 2 个字节，故其地址为 0xC00E。

各字节的存储分配如下表所示。

地址	0xC008	0xC009	0xC00A	0xC00B
内容	record.a (0x11)	record.a (0x01)	record.a (0x00)	record.a (0x00)
地址	0xC00C	0xC00D	0xC00E	0xC00F
内容	record.b	–	record.c	record.c

摩尔定律：1965 年，Intel 公司的创始人 Gordan Moore 根据当时的芯片技术(那时他们能够在一个芯片上制造有大约 64 个晶体管的电路)做出推断，预测在未来 10 年，芯片上的晶体管数量每年都会翻一番。这个预测有点乐观而且短视，事实是在超过 45 年时间中，半导体工业一直能够使这个数目每 18 个月翻一番

计算机内部的数据除了用二进制表示外，还有二进制编码的十进制数(一般采用 8421 码，因为只表示 0 到 9，所以这个编码中没有 1010 到 1111)

计算机中浮点数的表示法一般采用 IEEE 754 标准

IEEE 标准用$\mathrm{V} = (-1)^s \times M \times 2^E$的形式来表示一个数

符号　s 决定这个数是负数(s=1)还是正数(s=0)，而对于数值 0 的符号位解释作为特殊情况处理

尾数　M 是一个二进制小数，它的范围是$1 \sim 2 - 2^{-n}$

阶码　E 的作用是对浮点数进行加权，这个权重是 2 的 E 次幂

将浮点数的位模式划分为 3 段，分别对这些值编码：

一个单独的符号位 s 直接编码 s；

k 位的字段$\exp = e_{k-1} \dots e_1 e_0$编码阶码 E；

n 位小数字段$\mathrm{frac} = f_{n-1} \dots f_1 f_0$编码尾数 M，也依赖于阶码字段的值是否为 0

如果阶码的位模式既不全为 0 也不全为 1 时，称浮点数是规格化的

对于规格化数：

阶码字段被解释为以偏置值形式表示的有符号整数，即$\mathrm{E} = \mathrm{e} - \mathrm{Bias}$，其中 e 是$e_{k-1} \dots e_1 e_0$对应的无符号数，而 Bias 是等于$2^{k-1} - 1$的偏置值。尾数定义为$\mathrm{M} = 1 + \mathrm{frac}$

对于非规格化数：$\mathrm{E} = 1 - \mathrm{Bias}$　　$\mathrm{M} = \mathrm{frac}$

单精度表示：1+8+23=32 位；　双精度表示：1+11+52=64 位

单精度

31	30 … 23	22 … 0
s	exp	frac

双精度

63	62 … 52	51 … 32
s	exp	frac(51:32)

31 … 0
frac(31:0)

两个正的规格化浮点数，阶码大的那个大

计算机表示的数发生溢出的根本原因是机器的字长有限

左规：尾数算术左移 1 位，阶码减 1；右规：尾数算术右移 1 位，阶码加 1

浮点数溢出分为以下几种情况：

阶码上溢：正指数超过了最大允许值，有正上溢出和负上溢出

阶码下溢：负指数小于最小允许值，运算结果的绝对值小于机器所能表示的最小绝对值

尾数溢出：尾数最高有效位有进位(隐藏的符号位是 10 或 01)，此时要右规

非规格化尾数：尾数的最高数值位和符号位相同(00.0…或 11.1…)，此时要左规

浮点数的加减运算需要几个步骤：

对阶：使小数点的位置对齐，不断对阶码小的那个尾数做右规操作

尾数求和：按定点数的加法规则

规格化：为了防止溢出，需要进行左规或右规

舍入：浮点数的特有概念，在对阶和右规时可能丢失尾数低位影响精度，所以需要舍入，舍入不一定产生误差(移去一个 0 就直接舍去)，可能导致阶码上溢(移去一个 1 就在尾数的末尾加 1→可能又会使尾数溢出→再来一次右规→阶码加 1→阶码可能溢出)

计算机中最基本的加法单元是全加器，它有 3 个输入(2 个加数，1 个低位送过来的进位)和 2 个输出(1 个本位和，1 个准备送去高位的进位)

并行加法器的进位方式分为串行进位(进位信号逐级生成，速度由进位信号的传递时间决定)和并行进位(各级进位信号同时生成，速度快，缺点是电路结构复杂)

逻辑电路分为 2 类：时序逻辑电路(包含记忆单元，输出与之前的状态有关)，组合逻辑电路(输出只和输入有关)

ALU 是一种组合逻辑电路，由于加减乘除运算最后都可以归结为加法运算，所以 ALU 的核心是一个并行加法器

实际编程时较少采用堆排序，是因为堆排序比较的几乎都不是相邻元素，对 cache 极不友好。<u>数学上的时间复杂度不代表实际运行时的情况</u>

并行性是指多个任务在同一时刻同时处理，并发性是指多个事件在同一时间间隔内一个接一个地处理(也叫串行)

操作系统的基本特征包括并发(表面上并行，实际上并发，把处理器的运行时间分成多个时间片，作业排队完成。并行性需要硬件的支持，如多核处理器)、共享(进程之间资源共享，分为互斥共享方式和同时访问方式)、虚拟(就是给使用者提供假象，比如虚拟存储器)、异步(一个任务不是从开始到结束一直做下去，而是走走停停，在不同的任务之间不停切换)

操作系统作为计算机资源的管理者，它提供给应用程序的接口是系统调用(广义指令)，计算机开机后，它最终被加载到内存的系统区

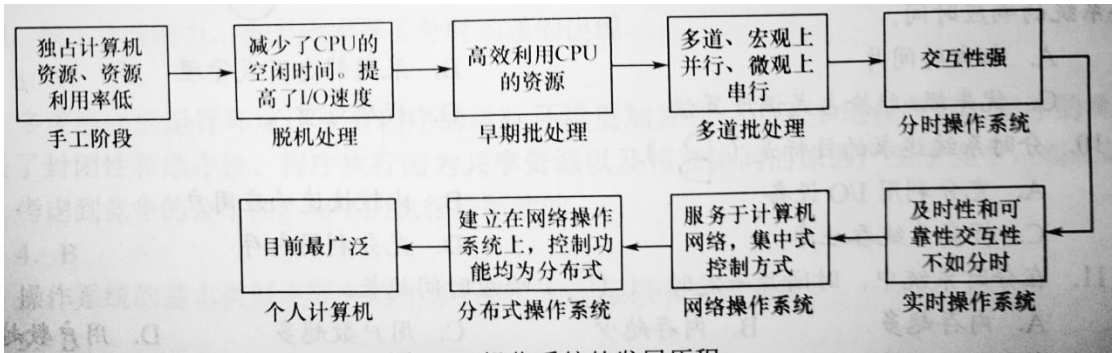


图 1-1　操作系统的发展历程

大多数操作系统内核包括 4 个方面的内容：时钟管理、中断机制(出现外中断或异常时，通过硬件实现，将计算机从用户态转到核心态，这是必须提供的功能)、原语(最接近硬件的部分，每个操作只能一气呵成)、系统控制的数据结构(如进程调度算法)

用户态：访管指令(用户请求使用特权指令，引发自愿的外中断)、系统调用的发生

核心态：CPU 可执行除访管指令外的所有指令、系统调用的执行、输入输出

操作系统管理内存：它只管数据放在哪里、哪里可以放、哪里不可以放、哪里有空闲

内部异常的检测由 CPU 的内部逻辑实现，处理后不一定会返回到发生异常的指令继续执行

子程序调用只需保存程序断点(该指令的下一条指令的地址，也就是 PC 的内容)；中断处理时最主要的两个寄存器是 PC 和 PSWR(程序状态字寄存器)，其中的内容都需要由中断隐指令来保存，其他通用寄存器中的内容则由操作系统保存

多道程序并发是指有的程序在 CPU 上运行的同时另一些任务正在 I/O 设备上传输。要实现这样的重叠必须要有中断机制和通道技术(一种硬件技术)的支持

系统调用命令被编译后，生成若干参数和陷入(Trap)指令，只有触发了 Trap，才能被认为是异常，从而转到核心态来执行

使用 C++实现复数乘法，例如 Input: "1+-1i", "1+-1i"　Output: "0+-2i"

```
    string complexNumberMultiply(string a, string b) {
        int ra,ia,rb,ib;
        char buff;
        stringstream aa(a),bb(b),ans;
        aa>>ra>>buff>>ia>>buff;
        bb>>rb>>buff>>ib>>buff;
        ans<<ra*rb-ia*ib<<"+"<<ra*ib+rb*ia<<"i";
        return ans.str();
    }
```

在字符串头部添加最少数目的字符使字符串变为回文，并返回这个最短的回文(kmp 算法)

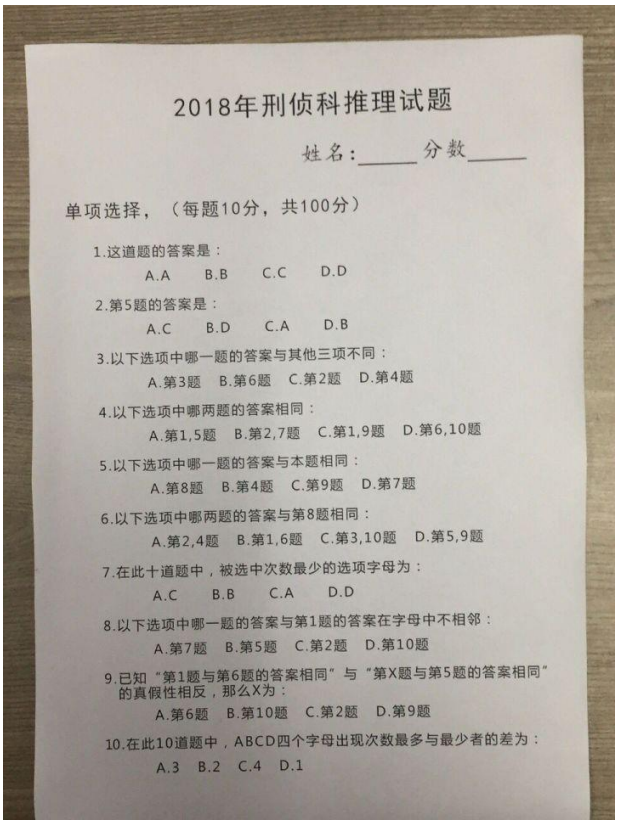
```
    string shortestPalindrome(string s) {
        string rev=s;
        reverse(rev.begin(),rev.end());
        string mirror=s+"#"+rev;
        vector<int> next(mirror.size(),0);
        for(int i=1;i<mirror.size();i++){
            int j=next[i-1];
            while(j>0&&mirror[i]!=mirror[j])
                j=next[j-1];
            next[i]=(j+=mirror[i]==mirror[j]);
        }
        return rev.substr(0,rev.size()-next[mirror.size()-1])+s; }
```

判断 N 个布尔值是否全部相等(Javascript 版本)：

```
return arr.indexOf(!arr[0])<0;
```

在 123456789 中插入+和-使得结果为 100(C 版本)：

```
#include<stdio.h>
#include<stdlib.h>
#define C10(x) if((x)==num)printf("%s = %d\n",#x,num);
#define C9(x) C10(x##9);C10(x+9);C10(x-9);
#define C8(x) C9(x##8);C9(x+8);C9(x-8);
#define C7(x) C8(x##7);C8(x+7);C8(x-7);
#define C6(x) C7(x##6);C7(x+6);C7(x-6);
#define C5(x) C6(x##5);C6(x+5);C6(x-5);
#define C4(x) C5(x##4);C5(x+4);C5(x-4);
#define C3(x) C4(x##3);C4(x+3);C4(x-3);
#define C2(x) C3(x##2);C3(x+2);C3(x-2);
int main(int argc,char **argv){
    int num;
    if(argc==1) num=100;
    else       num=atoi(argv[1]);
    C2(1);
    C2(-1);
    return 0;
}
```



2018年刑侦科推理试题

姓名:_____ 分数_____

单项选择，（每题10分，共100分）

1.这道题的答案是：
A.A B.B C.C D.D

2.第5题的答案是：
A.C B.D C.A D.B

3.以下选项中哪一题的答案与其他三项不同：
A.第3题 B.第6题 C.第2题 D.第4题

4.以下选项中哪两题的答案相同：
A.第1,5题 B.第2,7题 C.第1,9题 D.第6,10题

5.以下选项中哪一题的答案与本题相同：
A.第8题 B.第4题 C.第9题 D.第7题

6.以下选项中哪两题的答案与第8题相同：
A.第2,4题 B.第1,6题 C.第3,10题 D.第5,9题

7.在此十道题中，被选中次数最少的选项字母为：
A.C B.B C.A D.D

8.以下选项中哪一题的答案与第1题的答案在字母中不相邻：
A.第7题 B.第5题 C.第2题 D.第10题

9.已知“第1题与第6题的答案相同”与“第X题与第5题的答案相同”的真假性相反，那么X为：
A.第6题 B.第10题 C.第2题 D.第9题

10.在此10道题中，ABCD四个字母出现次数最多与最少者的差为：
A.3 B.2 C.4 D.1

程序输出结果：

```
123+45-67+8-9 = 100
123+4-5+67-89 = 100
123-45-67+89 = 100
123-4-5-6-7+8-9 = 100
12+3+4+5-6-7+89 = 100
12+3-4+5+67+8+9 = 100
12-3-4+5-6+7+89 = 100
1+23-4+56+7+8+9 = 100
1+23-4+5+6+78-9 = 100
1+2+34-5+67-8+9 = 100
1+2+3-4+5+6+78+9 = 100
-1+2-3+4+5+6+78+9 = 100
```

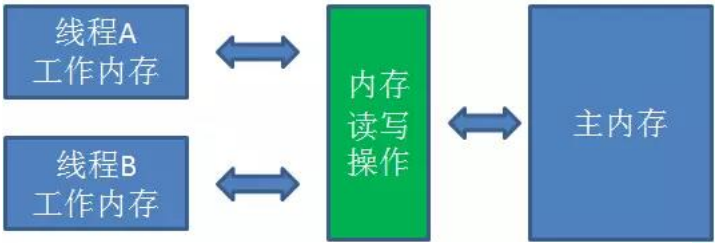
2018 年刑侦科推理试题暴力解法(C 版本)：

```
#include <stdio.h>
#include <stdlib.h>
#define A(k) ((i >> (k << 1)) & 3)
int main() {
    for (int i = 0; i < 1 << 20; i++) {
        int c[4] = { 0 }, cmin = 10, cmax = 0;
        for (int j = 0; j < 10; j++) c[A(j)]++;
        for (int j = 0; j < 4; j++) cmin = cmin < c[j] ? cmin : c[j];
        for (int j = 0; j < 4; j++) cmax = cmax > c[j] ? cmax : c[j];
        if ((A(1) == 0 && A(4) == 2 ||
             A(1) == 1 && A(4) == 3 ||
             A(1) == 2 && A(4) == 0 ||
             A(1) == 3 && A(4) == 1) &&
            (A(2) == 0 && A(2) != A(5) && A(2) != A(1) && A(2) != A(3) ||
             A(2) == 1 && A(5) != A(2) && A(5) != A(1) && A(5) != A(3) ||
             A(2) == 2 && A(1) != A(2) && A(1) != A(5) && A(1) != A(3) ||
             A(2) == 3 && A(3) != A(2) && A(3) != A(5) && A(3) != A(1)) &&
            (A(3) == 0 && A(0) == A(4) ||
             A(3) == 1 && A(1) == A(6) ||
             A(3) == 2 && A(0) == A(8) ||
             A(3) == 3 && A(5) == A(9)) &&
            (A(4) == 0 && A(7) == 0 ||
```

```
            A(4) == 1 && A(3) == 1 ||
            A(4) == 2 && A(8) == 2 ||
            A(4) == 3 && A(6) == 3) &&
           (A(5) == 0 && A(1) == A(7) && A(3) == A(7) ||
            A(5) == 1 && A(0) == A(7) && A(5) == A(7) ||
            A(5) == 2 && A(2) == A(7) && A(9) == A(7) ||
            A(5) == 3 && A(4) == A(7) && A(8) == A(7)) &&
           (A(6) == 0 && c[2] < c[0] && c[2] < c[1] && c[2] < c[3] ||
            A(6) == 1 && c[1] < c[0] && c[1] < c[2] && c[1] < c[3] ||
            A(6) == 2 && c[0] < c[1] && c[0] < c[2] && c[0] < c[3] ||
            A(6) == 3 && c[3] < c[0] && c[3] < c[1] && c[3] < c[2]) &&
           (A(7) == 0 && abs(A(0) - A(6)) != 1 ||
            A(7) == 1 && abs(A(0) - A(4)) != 1 ||
            A(7) == 2 && abs(A(0) - A(1)) != 1 ||
            A(7) == 3 && abs(A(0) - A(9)) != 1) &&
           (A(8) == 0 && (A(0) == A(5)) != (A(5) == A(4)) ||
            A(8) == 1 && (A(0) == A(5)) != (A(9) == A(4)) ||
            A(8) == 2 && (A(0) == A(5)) != (A(1) == A(4)) ||
            A(8) == 3 && (A(0) == A(5)) != (A(8) == A(4))) &&
           (A(9) == 0 && cmax - cmin == 3 ||
            A(9) == 1 && cmax - cmin == 2 ||
            A(9) == 2 && cmax - cmin == 4 ||
            A(9) == 3 && cmax - cmin == 1))
           for (int j = 0; j < 10; j++)
               putchar('A' + A(j));
    }
}
```

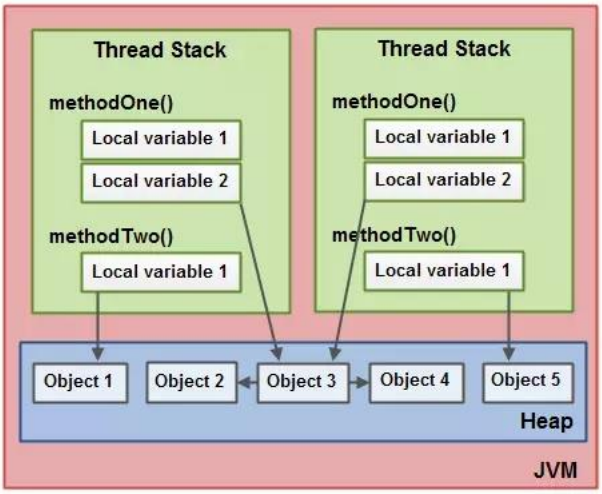
程序输出结果：

BCACACDABA



volatile 关键字在 Java 中的作用：保证对其他线程可见，当一个线程修改了该变量的值，新的值会立刻同步到主内存中。

Java 内存模型图解：



UNIX 操作系统下的僵尸进程：父进程还在运行，子进程挂了，都是父进程没有使用 wait()来清理子进程的信息，导致子进程虽然运行实体已经消失但仍在进程表中占据一项，导致资源浪费，这个子进程就是僵尸进程。杀掉父进程即可把这个僵尸进程变成孤儿进程，孤儿进程统一被 init 进程收养并被循环地 wait()，释放资源。

用 ps 命令查看时，状态为 Z 的进程是僵尸进程。

C++中智能指针最基本的概念是引用计数，指针内部的计数器记录了当前内存资源有多少指针正在引用，手动申请资源，计数器为 0 时自动释放资源。它与 JVM 垃圾回收机制的区别在于，JVM 基于 tracing GC，需要全局的对象图信息。内存越紧张时 tracing GC 的效率反而越低。

树状数组是一种用于维护数组前缀和信息的数据结构

```
#include<stdio.h>
#include<string.h>
int maxn=100,tree[101];
void add(int x,int v){        //单点修改
    for(;x<maxn;x+=x&-x)
        tree[x]+=v;
}
int sum(int x){               //区间查询
    int ans=0;
    for(;x>0;x-=x&-x)
        ans+=tree[x];
    return ans;
}
int findKthElement(int k){
    int lo=1,hi=maxn,mid;
    while(lo<hi){
        mid=(lo+hi)/2;
        if(sum(mid)>=k)
            hi=mid;
        else
            lo=mid+1;
    }
    return lo;
}
int main(void){
    int n,x;
    scanf("%d",&n);
    memset(tree,0,sizeof(tree));
    for(int i=0;i<n;i++){
        scanf("%d",&x);
        add(x,1);
        printf("  %d",sum(x-1));
    }
    printf("\n%d",findKthElement(3));
    return 0;
}
```

```
10
23
  0
12
  0
33
  2
24
  2
13
  1
56
  5
7
  0
8
  1
32
  6
19
  4
12
```

2016 的阶乘末尾第一个非零数字是几？

```
>>> print(str(eval('*'.join(map(str,range(1,2016))))).rstrip('0')[-1])
4
```

统计存放在硬盘中的 pdf 版费曼物理学讲义第三卷里面 1 多还是 0 多

```
import struct, os
filename, count_one, count_zero = 'feynman_chinese_book_3.pdf', 0, 0
for current_byte in list(open(filename, 'rb').read()):
    count_one += bin(struct.unpack("B", current_byte)[0]).count('1')
    count_zero = os.path.getsize(filename) * 8 - count_one
print 'One: ' + str(count_one) + 'times\nZero: ' + str(count_zero) +
'times\nOne/Zero: ' + str(float(count_one)/float(count_zero))
```

输出：

```
One: 53521103times
Zero: 55626937times
One/Zero: 0.962143628365
```

所有合法的中国象棋将帅摆法：

```
    int i=81;
```

```
    while(i--){
        if(i/9%3==i%9%3)
            continue;
        printf("A = %d, B = %d\n",i/9+1,i%9+1);
    }
```

从 1 写到 n 的所有正整数写过的 1 的个数（C++）：

```
    int countDigitOne(int n) {
        int ans=0;
        for(long long i=1;i<=n;i*=10){
            long long divider=i*10;
            ans+=(n/divider)*i+min(max(n%divider-i+1,0LL),i);
        }
        return ans;
}
```

数组中累计乘积小于 K 的连续子数组的个数（C）：

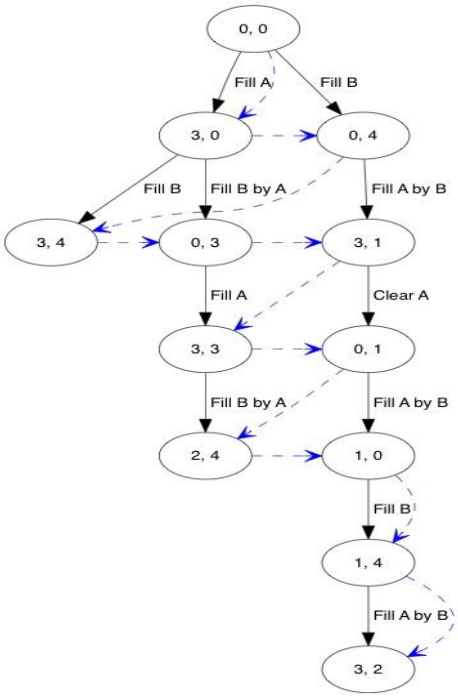
```
int numSubarrayProductLessThanK(int* nums, int numsSize, int k) {
    if(k<=1) return 0;
    int ans=0,left=0,prod=1;
    for(int right=0;right<numsSize;right++){
        prod*=nums[right];
        while(prod>=k)
            prod/=nums[left++];
        ans+=right-left+1;
    }
    return ans;
}
```

写 C++的时候如果懒得 include 一堆头文件(包括 STL) ,可以用#include<bits/stdc++.h>代替之。

两个杯子，一个盛 3 升水，一个盛 4 升水，怎样得到 2 升水(C 版本)

```
#include<stdio.h>
#define MIN(a,b) (a<b?a:b)
#define M 3        //如果写成 const int,就会报错"variably modified q at file scope"
#define N 4
#define G 2
typedef struct State{            //典型的状态空间搜索问题
    int a,b;                    //分别是 M=3 升水杯(A)和 N=4 升水杯(B)中的水量
    const struct State* parent;
    const char* action;         //表示合法状态转移的句子
}State;
State q[(M+1)*(N+1)];            //存储待展开的状态的队列
int front=0,back=0,visited[M+1][N+1];//队首队尾以及是否已经遍历过相同的状态
void Enqueue(int a,int b,const State* parent,const char* action){
    if(!visited[a][b]){
        State t={a,b,parent,action};
        q[back++]=t;
        visited[a][b]=1;
    }
}
void Backtrace(const State* s){ //找到目标状态后反向追踪其父状态直到最初状态
    if(s){
        Backtrace(s->parent);       //递归技巧反转打印次序
        printf("(%d, %d) %s\n",s->a,s->b,s->action);
    }
}
int main(void){
    Enqueue(0,0,NULL,"Initialize");     //初始状态
    while(front<back){                  //当队列非空
```

```
        const State* s=&q[front++];     //取出一个状态
        if(s->a==G || s->b==G){         //检查该状态是否达成目标
            Backtrace(s);
            printf("\n");
            continue;
        }                               //否则，尝试状态转移
        Enqueue(0,s->b,s,"Clear A");
        Enqueue(s->a,0,s,"Clear B");
        Enqueue(M,s->b,s,"Fill A");
        Enqueue(s->a,N,s,"Fill B");
        int t1=MIN(s->b,M-s->a),t2=MIN(s->a,N-s->b);
        Enqueue(s->a+t1,s->b-t1,s,"Fill A by B");
        Enqueue(s->a-t2,s->b+t2,s,"Fill B by A");
    }
    printf("Processed %d states\n",back);
}
```



输出：

```
(0, 0) Initialize
(3, 0) Fill A
(0, 3) Fill B by A
(3, 3) Fill A
(2, 4) Fill B by A

(0, 0) Initialize
(0, 4) Fill B
(3, 1) Fill A by B
(0, 1) Clear A
(1, 0) Fill A by B
(1, 4) Fill B
(3, 2) Fill A by B

Processed 12 states
```

打印圆周率前 800 位(C 版本)：

```
#include<stdio.h>
long a=10000,b,c=2800,d,e,f[2801],g;
int main(void){
    for(;b-c;)f[b++]=a/5;
    for(;d=0,g=c*2;c-=14,printf("%.4d",e+d/a),e=d%a)
    for(b=c;d+=f[b]*a,f[b]=d%--g,d/=g--,--b;d*=b);
}
```

判断一个数字中是否有一位是数字 5：

```
bool f(unsigned long long n){return n%10==5||n&&f(n/10);}
```

one-line tree in python:

```
In [1]: from collections import defaultdict
In [2]: def tree():return defaultdict(tree)
In [3]: players=tree()
In [4]: players['G2']['AWP']='KennyS'
In [5]: players['SK']['AWP']='Fallen'
In [6]: import json
In [7]: print(json.dumps(players))
{"SK": {"AWP": "Fallen"}, "G2": {"AWP": "KennyS"}}
```

Python 打印八皇后：

```
print [__import__('sys').stdout.write('\n'.join('.'*i+'Q'+'.'*(8-i-1)for
i in vec)
+ "\n=========\n")for vec in __import__('itertools').permutations(range(8))
      if 8 == len(set(vec[i]+i for i in range(8))) == len(set(vec[i]-i for
i in range(8)))]
```

打印 99 乘法表：

```
print '\n'.join([' '.join(['%s*%s=%-2s' % (y, x, x*y) for y in range(1, x+1)])
for x in range(1, 10)])
```

纯函数式的链表实现：

```
In [1]: def cons(x,y):return lambda m:m(x,y)
In [2]: def car(z):return z(lambda p,q:p)
In [3]: def cdr(z):return z(lambda p,q:q)
In [4]: a=cons(1,cons(2,cons(3,4)))
In [5]: car(a)
Out[5]: 1
In [6]: car(cdr(cdr(a)))
Out[6]: 3
```

判断字符串 s 是否是字符串 t 的子序列：

```
bool isSubsequence(char *s,char *t){
    while(*t)
        s+=*s==*t++;
    return !*s;
}
```

删除字符串字母中的星号：

```
#include<stdio.h>
#include<string.h>
void RemoveAsterisks(char* str){
    char *begin,*end,*out;
    for(begin=str;*begin=='*';++begin);     //找星号前缀
    if(!*begin)                            //如果全是星号
        return;                            //直接返回原值
    for(end=str+strlen(str);end[-1]=='*';--end);    //找星号后缀
    for(out=begin;begin!=end;++begin)       //过滤[begin,end)中非星号的字符
        if(*begin!='*')
            *out++=*begin;                 //输出至 out
    for(;*end!='\0';++end)                  //加上星号后缀
        *out++=*end;
    *out++='\0';                           //加上终止符
}
int main(void){
    char str[]="****K**AB*S*HW*HS*A**S***Z****IIO*W**";
    RemoveAsterisks(str);
    puts(str);                 //****KABSHWHSASZIIOW**
}
```

打印字符的叠筐：

```
#include<stdio.h>
int main(void){
    int outputbuf[82][82];          //输出缓存
    char a,b;
    int n;
    int firstcase=1;    //是否为第一组输入数据
    while(scanf("%d %c %c",&n,&a,&b)==3){
        if(firstcase==1)
```

```
            firstcase=0;
        else
            printf("\n");
        for(int i=1,j=1;i<=n;i+=2,j++){ //从内到外输出每个圈
            int x=n/2+1,y=x;
            x-=j-1;y-=j-1;          //每个圈右上角点的坐标
            char c=j%2==1?a:b;      //当前圈要用哪个字符
            for(int k=1;k<=i;k++){    //当前圈  左、上、右、下
                outputbuf[x+k-1][y]=c;
                outputbuf[x][y+k-1]=c;
                outputbuf[x+i-1][y+k-1]=c;
                outputbuf[x+k-1][y+i-1]=c;
            }
        }
        if(n!=1){               //四角置空格
            outputbuf[1][1]=' ';
            outputbuf[n][1]=' ';
            outputbuf[1][n]=' ';
            outputbuf[n][n]=' ';
        }
        for(int i=1;i<=n;i++){
            for(int j=1;j<=n;j++)
                printf("%c",outputbuf[i][j]);
            printf("\n");
        }
    }
}
```

```
11 B A
 AAAAAAAAA
ABBBBBBBBBA
ABAAAAAAABA
ABABBBBBABA
ABABAAABABA
ABABABABABA
ABABAAABABA
ABABBBBBABA
ABAAAAAAABA
ABBBBBBBBBA
 AAAAAAAAA
7 $ %

 %%%%%
%$$$$$%
%$%%%$%
%$%$%$%
%$%%%$%
%$$$$$%
 %%%%%
```

n 位无符号和带符号整数的加减运算都可以在 n 位加法器中实现

带符号整数加减运算的溢出判断规则是：加法器的两个输入端的符号相同且不同于输出端的符号，或者，加法器完成一步加法操作时次高位（最高数字位）的进位和最高位（符号位）的进位不同。

存取时间：从存储器读出或写入一次信息所需的平均时间

存取周期：连续两次访问存储器之间所必需的最短时间间隔

访问存储单元的 3 种方式：随机（存取时间与存储单元的物理位置无关，分为 RAM 和 ROM，ROM 的写入速度远远慢于读取速度，又称为只读存储器，因为只读，保证了数据的非易失性，并且不需要刷新，而 RAM 是易失性的。ROM 一般用作 U 盘、固态硬盘，这两个东西都是基于闪存技术）、顺序（存取时间与存储单元的物理位置严格相关，如磁带、光盘 CD-ROM，注意它并不属于 ROM）、直接（介于随机访问和顺序访问之间，如磁盘）

存储系统中，按照 CPU—Cache—内存—外存的顺序，相邻的两两之间都可以相互交换信息，唯一的不相邻是，CPU 和内存也可以直接交换信息

“Cache—内存”解决 CPU 和内存速度不匹配的问题，“内存—外存”解决存储的总体容量问题

为了提高 CPU 访问存储器的速度，采用多端口（空间并行）或多模块（时间并行，分为高位和低位，高位是连续存放，低位是交叉存放，低位符合局部性原理）

CPU 要访问的信息已经在 Cache 中的比率称为 Cache 的命中率，所以很明显，访问内存的次数=未命中的次数。一个程序从存储器中读取数据的速率叫做读带宽，读带宽的时间和空间局部性的二维函数的图像称为存储器山。

RAM 分为静态(S)的和动态(D)的

	SRAM	DRAM
存储单元	两个稳态的六晶体管电路	对电容的充电放电
信息被读出时是否发生改变（破坏性读出）	否	是
刷新(每隔一段时间按原来的内容写入)	不要	要 (一次占用一个存取周期)
传输存储单元行列的地址	同时	分两次
速度	快	慢
集成度	低	高
发热量	大	小
成本	高	低
主要用途	Cache	内存

由于 Cache 中存放的是内存中一部分信息的复制，所以存储系统的总容量不能认为是内存容量和 Cache 容量的简单相加

当 CPU 访问内存时，先要把内存物理地址发给 Cache 查看该地址是否在 Cache 中

Cache 和内存的 3 种映射方式：

直接映射：内存块和 Cache 块一一对应，无条件替换。地址结构（对 CPU 发送的内存物理地址的拆分方式）为：内存块标记（tag）+Cache 块地址+块内地址

全相联映射：可以把内存块装入 Cache 中任何位置，

地址结构为：内存块标记（tag）+块内地址

组相联映射：把 Cache 分组，组间直接映射，组内全相联映射，

地址结构：内存块标记（tag）+组地址+块内地址

块内地址位数=log2（块内字节数）

Cache 块地址位数=log2（Cache 容量÷块容量）　　转成十进制意思是 Cache 中第几块

Tag 位数=log2（内存容量÷Cache 容量）

组地址位数=log2（Cache 容量÷块容量÷组相联的路数）=log2 组数

某个存储器中地址长度（位数）=log2（该存储器的容量）

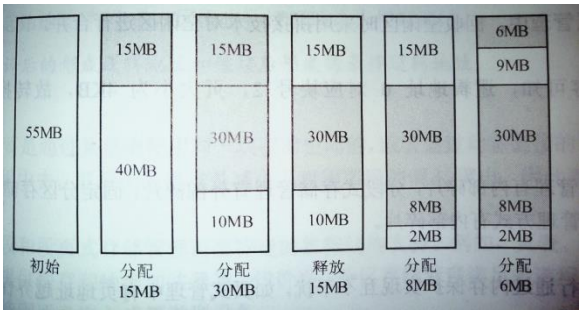
注意：表示存储器容量都默认用字节(B)做单位

Cache 写命中（要被修改的单元在 Cache 中）有两种处理方式：直写法（write-through，在 Cache 中修改之后马上在内存中也修改，增加了访问次数，每次写都会引起总线流量）、写回法（write-back，只有当替换算法要驱逐更新过的块时，才把它写到内存中，需要给每一个 Cache 块一个 dirty bit，表示这个块是否被修改过）

计算机对用户程序的 3 个处理步骤：编译、链接、装载。链接分为静态（链接成一个完整的.exe 之后就不再拆开）和动态的，装载分为绝对（编程阶段就把物理地址计算好）、静态重定位（装载时把逻辑地址转为物理地址，但装入后不能改变）、动态重定位（要执行时再决定装入的地址，装入后也可能换出）

逻辑地址+重定位（基址）寄存器中的值=物理地址。然后和界地址寄存器中的值比较，判断是否越界。实际工作时为了节省内存空间采用的两种方法是覆盖（打破了必须把一个进程的全部信息装入内存后才能运行的限制，已经成为历史）、交换（不同的进程之间不断切换）

内存分配方式：连续（单一、固定分区、动态分区）、非连续（页式、段式、段页式）
其中连续的动态分区有两种分配策略（选择空闲区的策略）：首次适应（按照地址顺序找到第一个大小满足要求的空闲区）、最佳适应（按照容量递增顺序找到第一个大小满足要求的空闲区）



3 种非连续分配方式的特点
页式：无外部碎片。内存中的是慢表，Cache 中的是快表(TLB)，快表是慢表的一个小副本。为了避免把全部页表一直放在内存中，引入多级页表，缺点是会增加缺页(不命中)的次数。
页大小固定，地址空间是单一线性的。页目录表就是页表，其中一些页用来存放页表项
每个进程有一个页表，并且常驻内存，
页表项数目×页面大小=逻辑地址数目　　页表项数目×页表项大小=页表最大占用空间
页内偏移量位数=log2（页面大小）
页目录号=(((unsigned int)(逻辑地址))>>(页表索引位数+页内偏移量位数))&(目录号位数个的 1)
段式：无内部碎片。段长度不固定，地址空间是二维的
段页式：用户地址空间分段，物理存储空间分页。取一次数据先去内存查段表，再去内存查页表，重定位出物理地址后再访问内存，一共要 3 次
无论系统采用什么内存管理方式，访问内存都是以字节或字为单位。系统提供给用户的空间大小为总的空间大小减去段表和页表的长度
当系统采用虚拟存储器时，作业在运行前不必全部装入内存，且在运行过程中也不必一直驻留内存
Cache 解决速度问题，虚拟存储器解决容量问题。虚拟存储器需要非连续内存分配技术的支持，虚存缺页对系统性能影响更大
有时也在数字末尾用 H 表示十六进制，B 表示二进制
虚拟地址=虚页号+页内地址　　物理地址=页框号+页内地址
虚页太大或太小都会减慢操作速度，工作（驻留）集是指某段时间内进程要访问的页面集合
决定应该换入哪页换出哪页就是虚存的页面置换策略，，有以下几种：
最佳（OPT，淘汰之后最长时间内不会再被访问的页面，无法实现）
先进先出（FIFO，淘汰驻留内存时间最长的页面，可能发生分配的实页数目增加而缺页数反而增加的异常，称为 belady 异常）
最近最久未使用（LRU，淘汰最近最长时间未访问过的页面）

访问页面	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
物理块 1	7	7	7	2		2		4	4	4	0			1		1		1		
物理块 2		0	0	0		0		0	0	3	3			3		0		0		
物理块 3			1	1		3		3	2	2	2			2		2		7		
缺页否	√	√	√	√		√		√	√	√	√			√		√		√		

时钟置换（CLOCK，给每一页一个使用位，循环扫描）
抖动（Thrashing）现象是指刚换出的页很快又要被访问，从而导致的频繁地页面置换。

磁盘交换区用于存放从内存换出的进程。发生抖动现象的根本原因是页面置换策略的不合理，撤销部分进程可以防止抖动，磁盘交换区和进程优先级都与抖动现象无关

在请求分页系统中，页面分配策略与页面置换策略不能组合使用的是固定分配和全局置换，因为页面数不变

段式分配将程序按逻辑划分，有利于动态链接

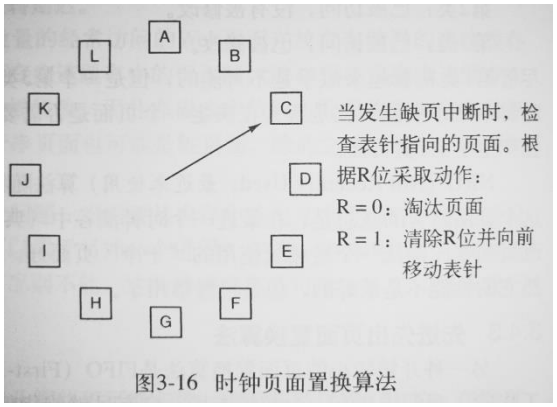


图3-16 时钟页面置换算法

应该采用的页表级数=（虚拟地址位数-页内偏移位数）÷（log2 一页内能放下的页表项数）

文件控制块（FCB，也叫文件目录项）。系统打开文件的操作是将该文件的 FCB 存入内存的活跃文件目录表（目录文件）。文件的逻辑结构是用户看到的，文件的物理结构是计算机实现的。按照逻辑结构，文件分为无结构（流式）和有结构（记录式）的。按照物理结构（实现方式，即存储空间的分配方式），文件分为顺序、链接、索引分配的。

	访问第 n 个记录需要访问磁盘的次数	优点	缺点
顺序分配	1	顺序存取时速度快，文件定长时可以随机访问	产生碎片，不利于文件的修改与扩充
链接分配	n	解决了外存的碎片问题	只能按照文件指针链顺序访问，指针占用空间
索引分配	m 级索引需要 m+1	可以随机访问，利于文件的增删	索引表占用空间，且它的查找策略影响文件系统效率

为了管理磁盘的空闲块，通常用一个位图，用二进制的一位来表示对应的块是否空闲

对一个文件的访问，由用户访问权限和文件属性共同限制。read 系统调用只需要 open 系统调用返回的文件描述符，不需要文件名为参数。

操作系统通过文件链接方式实现文件的共享，分为硬链接和软（符号）链接。当建立软链接时，链接出来的文件直接复制原文件的引用计数值，删除掉的原文件不影响链接文件的引用计数值。当建立硬链接时，链接文件和原文件的引用计数值都加 1，删除原文件后链接文件的引用计数值减 1，如果原文件的引用计数值不为 0，则不能删除原文件。

磁盘上的同心圆就是一个个的磁道，磁道分成一块块的就是扇区，一个扇区就是一个数据块由于扇区按照固定的圆心角划分，所以密度从最外面的磁道到最里面的磁道增加，磁盘的存储能力受限于最里面的磁道的最大记录密度

一次完整的磁盘读/写操作：

半径方向移动磁头（寻道时间，分为启动磁头的时间+跨越 n 条磁道的时间）→圆弧方向旋转磁盘（延迟时间，平均值=$\frac{1}{2r}$，r 为磁盘每一秒转过的扇区数目）→读写数据（传输时间，=$\frac{b}{rN}$，b 是每次要读/写的字节数，N 是一个扇区上的字节数）

磁盘的调度策略直接决定了寻道时间，有以下 4 种：

先来先服务（FCFS，根据进程请求访问的先后顺序来移动磁头）
最短寻找时间优先（SSTF，选择当前距离磁头最近的磁道，可能会产生"饥饿"现象）
电梯（SCAN，在 SSTF 的基础上规定了磁头的移动方向）
循环扫描(C-SCAN ,磁头只需要移动到最远处的一个请求即可返回 ,不需要到达磁盘端点）
磁盘是可共享设备（分时共享），是指某一段时间内可以有多个作业访问，但某一时刻只能有一个作业访问。

设文件索引结点中有 7 个地址项，其中 4 个地址项是直接地址索引，2 个地址项是一级间接地址索引，1 个地址项是二级间接地址索引，每个地址项大小为 4B，若磁盘索引块和磁盘数据块大小均为 256B，则可表示的单个文件最大长度是（ C ）。

A. 33KB　　B. 519KB　　C. 1057KB　　D. 16516KB

2. C

每个磁盘索引块和磁盘数据块大小均为 256B，每个磁盘索引块有 256/4=64 个地址项。因此，4 个直接地址索引指向的数据块大小为 4×256B；2 个一级间接索引包含的直接地址索引数为 2×(256/4)，即其指向的数据块大小为 2×(256/4)×256B。1 个二级间接索引所包含的直接地址索引数为(256/4)×(256/4)，即其所指向的数据块大小为(256/4)×(256/4) ×256B。即 7 个地址项所指向的数据块总大小为 4×256+2×(256/4)×256+(256/4)×(256/4)×256=1082368B=1057KB。

在文件的索引节点中存放直接索引指针 10 个，一级和二级索引指针各 1 个。磁盘块大小为 1KB，每个索引指针占 4 个字节。若某文件的索引节点已在内存中，则把该文件偏移量（按字节编址）为 1234 和 307400 处所在的磁盘块读入内存，需访问的磁盘块个数分别是（ B ）。

A. 1,2　　B. 1,3　　C. 2,3　　D. 2,4

10 个直接索引指针指向的数据块大小为 10*1KB=10KB；
每个索引指针占 4B，则每个磁盘块可存放 1KB/4B=256 个索引指针，
一级索引指针指向的数据块大小为：256*1KB=256KB，
二级索引指针指向的数据块大小为：256*256*1KB=2^{16}KB=64MB。
按字节编址，偏移量为 1234 时，因 1234B＜10KB，则由直接索引指针可得到其所在的磁盘块地址。文件的索引节点已在内存中，则地址可直接得到，故仅需 1 次访盘即可。
偏移量为 307400 时，因 10KB+256KB＜307400B＜64MB，可知该偏移量的内容在二级索引指针所指向的某个磁盘块中，索引节点已在内存中，故先访盘 2 次得到文件所在的磁盘块地址，再访盘 1 次即可读出内容，故共需 3 次访盘。

1）根据页式管理的工作原理，应先考虑页面大小，以便将页号和页内位移分解出来。页面大小为 4KB，即 2^{12}，则得到页内位移占虚地址的低 12 位，页号占剩余高位。可得三个虚地址的页号 P 如下（十六进制的一位数字转换成 4 位二进制，因此，十六进制的低三位正好为页内位移，最高位为页号）：

2362H：P=2，访问快表 10ns，因初始为空，访问页表 100ns 得到页框号，合成物理地址后访问主存 100ns，共计 10ns+100ns+100ns=210ns。

1565H：P=1，访问快表 10ns，落空，访问页表 100ns 落空，进行缺页中断处理 10^8ns，访问快表 10ns，合成物理地址后访问主存 100ns，共计 10ns+100ns+10^8ns+10ns+100ns=100 000 220ns。

25A5H：P=2，访问快表，因第一次访问已将该页号放入快表，因此花费 10ns 便可合成物理地址，访问主存 100ns，共计 10ns+100ns=110ns。

2）当访问虚地址 1565H 时，产生缺页中断，合法驻留集为 2，必须从页表中淘汰一个页面，根据题目的置换算法，应淘汰 0 号页面，因此 1565H 的对应页框号为 101H。由此可得 1565H 的物理地址为 101565H。 ?

页面大小为 4KB，一次内存的访问时间是 100ns，一次快表（TLB）的访问时间是 10ns，处理一次缺页的平均时间 10^8ns（已含更新 TLB 和页表的时间），进程的驻留集大小固定为 2，采用最近最少使用置换算法（LRU）和局部淘汰策略。假设①TLB 初始为空；②地址转换时先访问 TLB，若 TLB 未命中，再访问页表（忽略访问页表之后的 TLB 更新时间）；③有效位为 0 表示页面不在内存，产生缺页中断，缺页中断处理后，返回到产生缺页中断的指令处重新执行。设有虚地址访问序列 2362H、1565H、25A5H，请问：

表 3-19　页表内容

页号	页框（Page Frame）号	有效位（存在位）
0	101H	1
1	—	0
2	254H	1

1）依次访问上述三个虚拟地址，各需多少时间？给出计算过程。

2）基于上述访问序列，虚地址 1565H 的物理地址是多少？请说明理由。

某计算机的主存地址空间大小为 256MB，按字节编址。指令 Cache 和数据 Cache 分离，均有 8 个 Cache 行，每个 Cache 行大小为 64B，数据 Cache 采用直接映射方式。现有两个功能相同的程序 A 和 B，其伪代码如下所示：

```
程序 A:
int a[256][256];
……
int sum_array1()
{
    int i,j,sum=0;
    for(i=0;i<256;i++)
        for(j=0;j<256;j++)
            sum += a[i][j];
    return sum;
}
```

```
程序 B:
int a[256][256];
……
int sum_array2()
{
    int i,j,sum=0;
    for(j=0;j<256;j++)
        for(i=0;i<256;i++)
            sum += a[i][j];
    return sum;
}
```

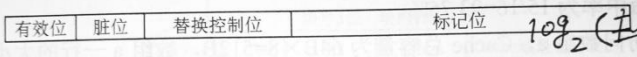
假定 int 类型数据用 32 位补码表示，程序编译时，i、j 和 sum 均分配在寄存器中，数组 a 按行优先方式存放，其首地址为 320（十进制数）。请回答下列问题，要求说明理由或给出计算过程。

1）若不考虑用于 Cache 一致性维护和替换算法的控制位，则数据 Cache 的总容量为多少？

2）数组元素 a[0][31]和 a[1][1]各自所在的主存块对应的 Cache 行号分别是多少（Cache 行号从 0 开始）？ 3　0

3）程序 A 和 B 的数据访问命中率各是多少？哪个程序的执行时间更短？

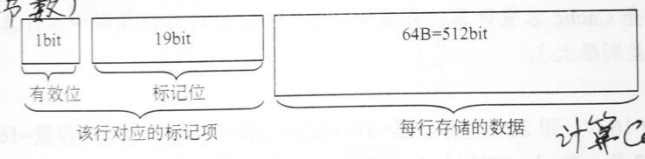
1）每个 Cache 行对应一个标记项，如下图所示。



$\log_2$(主存地址空间大小)

不考虑用于 Cache 一致性维护和替换算法的控制位。地址总长度为 28 位（2^{28}=256M），块内地址 6 位（2^6=64），Cache 块号 3 位（2^3=8），故 Tag 的位数为 28−6−3=19 位，还需使用一个有效位，故题中数据 Cache 行的结构如下图所示。

$\log_2$(Cache 行内字节数)

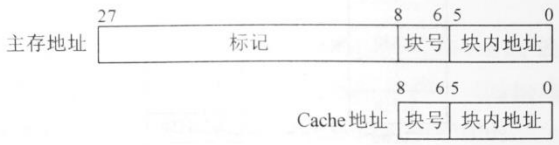


计算Cache总容量要注意

数据 Cache 共有 8 行，因此数据 Cache 的总容量为 8×(64+20/8)B=532B。

数组按行优先方式存放，首地址为 320，数组元素占 4 个字节。a[0][31]所在的主存块对应的 Cache 行号为$[(320+(0\times256+31)\times4\text{div}2^6)]\text{mod}2^3=6$；a[1][1]所在的主存块对应的 Cache 行号为$[(320+(1\times256+1)\times4)\text{div}2^6]\text{mod}2^3=5$。

另解：由 1）可知主存和 Cache 的地址格式如下图所示。



数组按行优先方式存放，首地址 320，数组元素占 4 个字节。a[0][31]的地址为 320+31×4=1 1011 1100_B，故其对应的 Cache 行号为 110_B=6；a[1][1]的地址为 320+256×4+1×4=1348=101 0100 0100_B，故其对应的 Cache 行号为 101_B=5。

3）数组 a 的大小为 256×256×4B=2^{18}B，占用 $2^{18}/64=2^{12}$ 个主存块，按行优先存放，程序 A 逐行访问数组 a，共需访问的次数为 2^{16} 次，未命中次数为 2^{12} 次（即每个字块的第一个数未命中），因此程序 A 的命中率为$(2^{16}-2^{12})/2^{16}\times100\%=93.75\%$。

另解：数组 a 按行存放，程序 A 按行存取。每个字块中存放 16 个 int 型数据，除访问的第一个不命中，随后的 15 个全都命中，访问全部字块都符合这一规律，且数组大小为字块大小的整数倍，故程序 A 的命中率为 15/16=93.75%。

程序 B 逐列访问数组 a，Cache 总容量为 64B×8=512B，数组 a 一行的大小为 1KB，正好是 Cache 容量的 2 倍，可知不同行的同一列数组元素使用的是同一个 Cache 单元，故逐列访问每个数据时，都会将之前的字块置换出，也即每次访问都不会命中，命中率为 0。

由于从 Cache 读数据比从主存读数据快很多，所以程序 A 的执行比程序 B 快得多。

计算机使用总线结构便于增减外设，同时减少了信息传输线的条数，但相对于专线结构，降低了信息传输的并行性和速度。总线的主设备是指获得总线控制权的设备，信息传输由主设备启动，主设备是"总线忙"信号的建立者，一条总线上可以有多个主设备，但在某一时刻只能由一个主设备控制总线，总线包括数据线、地址线（只能单向传输）、控制线

一张图片的信息量=像素×（log2 色数），单位是 bit

总线之间的通信采用同步方式（统一的与设备无关的时钟信号）和异步方式（传送双方相互的握手信号，分为不互锁、半互锁、全互锁。分别对应的握手信号是↘、↘↗、↘↗↘）

总线标准的英文缩写：

即插即用：PCI　USB　　系统：ISA　EISA　　设备：USB　　通信：RS-232C

串行：USB　RS-232C　　局部：PCI　VESA　AGP　PCI-Express

总线仲裁解决多个主设备竞争控制权的问题，分为链式、计数器定时、独立请求方式

I/O 设备通过设备控制器与主板的系统总线相连，I/O 指令是 CPU 系统指令的一部分，格式与其他通用指令不同。CPU 对外部设备的控制方式按照 CPU 的介入程度，从小到大分别为 I/O 通道方式、直接存储器访问（DMA）、中断方式、程序直接控制

通道程序放在内存里面，由通道从内存中取出来并执行

键盘、鼠标等输入设备一般都采用中断方式来实现，因为 CPU 需要及时响应这些操作。

I/O 端口是指接口电路中可以被 CPU 直接访问的寄存器，要能够被访问必须有一个端口地址，这个地址有两种编法：统一（当做存储器单元，CPU 用访存指令访问 I/O 端口，用不同的地址码来区分）、独立（在具有专门 I/O 指令的计算机中才可以这样做，靠不同的指令来区分）

中断向量就是中断服务程序的入口地址

中断优先级由高到低为：访管→程序性→重新启动。Cache 属于存储设备，不能提出中断请求，浮点数下溢可以当做机器 0 处理，不需要中断，而外部事件和浮点数上溢都会引起中断

设置中断屏蔽标志可以改变多个中断服务程序执行完的次序

右图中的硬件就是 CPU 完成的 3 步工作，CPU 响应中断时由硬件自动保存 PC 和 PSW 的状态

能产生 DMA 请求的设备只能是具有 DMA 接口的设备

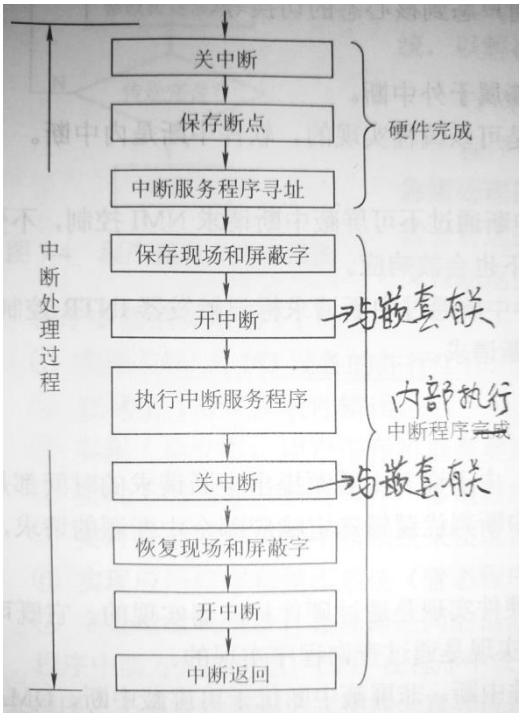


图 7-6　可嵌套中断的处理流程

	中断方式	DMA 方式
CPU 与外设	并行	并行
传送与主程序	并发	
优先级	低	高
是否要保护现场	是	否
请求什么	CPU 处理时间	总线使用权
在什么时候响应	指令执行完之后	总线事务完成之后
数据传送靠什么	软件	硬件
传送什么	字（每一个都要经过 CPU）	数据块

共享设备必须是可寻址的和可随机访问的设备，磁盘设备的 I/O 控制主要是采取 DMA 方式因为 DMA 用于块设备。系统为每一类设备配一个设备驱动程序

用户程序发出磁盘 I/O 请求后，系统的处理流程是：用户程序（用户级 I/O 软件）→系统调用处理程序（设备无关软件）→设备驱动程序→中断处理程序

控制顺序：通道→设备控制器（控制寄存器+状态寄存器+控制命令）→设备

I/O 子系统是操作系统内核的一部分，它包括缓冲区服务、SPOOLing（假脱机）技术等

假如：T：数据从磁盘到缓冲区的时间，M：数据从缓冲区到用户区的时间，C：CPU 对这一数据的处理时间

单缓冲区处理一块数据用时 max(C,T)+M　　双缓冲区处理一块数据用时 max(C+M,T)

在磁盘上设置多个分区不会改善 I/O 性能，反而会增加复杂性

在鼠标移动时，如果有高优先级的操作产生，为了记录鼠标活动的情况，必须使用缓冲技术

SPOOLing 技术：将独占设备改造为共享设备，用空间换时间，需要磁盘和内存中的缓冲区

设备独立性是指用户程序中只指明 I/O 使用的设备类型即可，与具体设备无关

一条指令是一组有意义的二进制位，它分为地址码段和操作码段，如果你要扩展操作码，指令的长度是不会变的。零地址的运算类指令又称堆栈运算指令，参与的两个操作数来自栈顶和次栈顶单元，堆栈指针 SP 是一个寄存器

数据寻址：确定本条指令的数据的地址。10 种数据寻址方式：(假设 A 是地址码，Ri 是寄存器，(X)表示 X 这个地址上放着的东西，寄存器也是一个地址，寄存器里面要放东西。BR 是基址寄存器，IX 是变址寄存器)

数据寻址方式	真实地址	访问内存次数
隐含	由程序来决定	0
立即	A 就是操作数	0
直接	A	1
一次间接	(A)	2
寄存器	Ri	0
寄存器一次间接	(Ri)	1
相对	(PC)+A	1
基址	(BR)+A BR 不变，A 可变	1
变址	(IX)+A IX 可变，A 不变。用于实现数组	1

一个计算机采用 32 位的指令，地址码为 12 位，如果已经定义了 250 条二地址指令，还可以定义多少条单地址指令？

二地址指令的操作码长度为 32-12-12=8，已经有了 250 条二地址指令，$2^8-250=6$，还可以定义 6×2^{12}条单地址指令

隐地址不给出明显的操作数地址，而是在指令中隐含操作数的地址，可以简化地址结构

指令寻址分为顺序（就是 PC 加 1）和跳跃（修改 PC 的值，但还是由 PC 指出地址，可以实现程序的条件和无条件转移，如果转移失败就又 PC+1）

无符号整数比较大于时转移的转移条件是 CF 和 ZF 都为 0，因为如果 A 大于 B，A-B 一定没有进位/借位（CF），也不为 0（ZF）

	CISC	RISC
定义	复杂指令系统	精简指令系统
应用	X86 架构	ARM、MIPS 架构
指令字长	不固定	定长
指令数目	≥200	＜100
各指令执行时间	相差很大	绝大多数在一个周期内
各指令使用频率	相差很大	都比较常用
控制方式	微程序	组合逻辑
指令流水线	可以实现	必须实现
通用寄存器数目	少	多
单条指令的功能	增强了	简化了

43.（11 分）某计算机字长为 16 位，主存地址空间大小为 128KB，按字编址。采用单字长指令格式，指令各字段定义如图 B-4 所示。

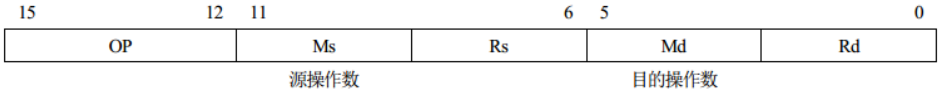


图 B-4

转移指令采用相对寻址方式，相对偏移量用补码表示，寻址方式定义见表 B-1。

表 B-1

Ms/Md	寻址方式	助记符	含义
000B	寄存器直接	Rn	操作数=(Rn)
001B	寄存器间接	(Rn)	操作数=((Rn))
010B	寄存器间接、自增	(Rn)+	操作数=((Rn))，(Rn)+1→Rn
011B	相对	D(Rn)	转移目标地址=(PC)+(Rn)

注：（X）表示存储器地址 X 或寄存器 X 的内容。

请回答下列问题：

1）该指令系统最多可有多少条指令？该计算机最多有多少个通用寄存器？存储器地址寄存器（MAR）和存储器数据寄存器（MDR）至少各需要多少位？

2）转移指令的目标地址范围是多少？

3）若操作码 0010B 表示加法操作（助记符为 add），寄存器 R4 和 R5 的编号分别为 100B 和 101B，R4 的内容为 1234H，R5 的内容为 5678H，地址 1234H 中的内容为 5678H，地址 5678H 中的内容为 1234H，则汇编语言为“add(R4),(R5)+”（逗号前为源操作数，逗号后为目的操作数）对应的机器码是什么（用十六进制表示）？该指令执行后，哪些寄存器和存储单元中的内容会改变？改变后的内容是什么？

1）操作码占 4 位，则该指令系统最多可有 2^4=16 条指令；操作数占 6 位，寻址方式占 3 位，于是寄存器编号占 3 位，则该机最多有 2^3=8 个通用寄存器；主存容量为 128KB，按字编址，计算机字长为 16 位，划分为 128KB/2B=2^{16} 个存储单元，故 MDR 和 MAR 至少各需 16 位。

2)PC 和 Rn 可表示的地址范围均为 0～2^{16}−1，而主存地址空间为 2^{16}，故转移指令的目标地址范围为 0000H～FFFFH（0～2^{16}−1）。

3）汇编语句“add (R4), (R5)”，对应的机器码为 0010 0011 0001 0101B=2315H。

该指令执行后，寄存器 R5 和存储单元 5678H 的内容会改变。执行后，R5 的内容从 5678H 变成 5679H。存储单元 5678H 中的内容变成该加法指令计算的结果 5678H+1234H=68ACH。

44.（14 分）某计算机采用 16 位定长指令字格式，其 CPU 中有一个标志寄存器，其中包含进位/借位标志 CF、零标志 ZF 和符号标志 NF。假定为该机设计了条件转移指令，其格式如下：

15　　11	10	9	8	7　　0
00000	C	Z	N	OFFSET

其中，00000 为操作码 OP；C、Z 和 N 分别为 CF、ZF 和 NF 的对应检测位，某检测位为 1 时表示需检测对应标志位，需检测的标志位中只要有一个为 1 就转移，否则不转移，例如，若 C=1，Z=0，N=1，则需检测 CF 和 NF 的值，当 CF=1 或 NF=1 时发生转移；OFFSET 是相对偏移量，用补码表示。转移执行时，转移目标地址为(PC)+2+2×OFFSET；顺序执行时，下条指令地址为(PC)+2。请回答下列问题。

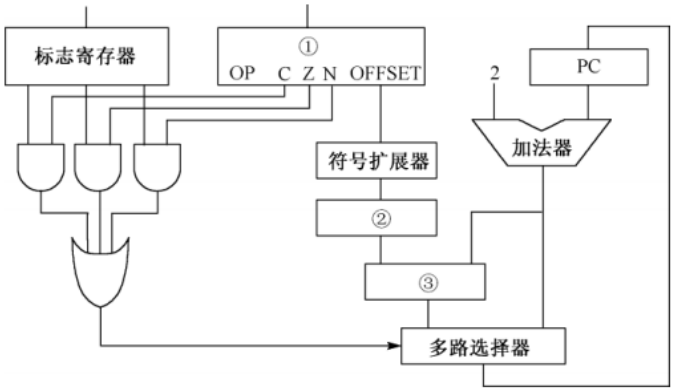
（1）该计算机存储器按字节编址还是按字编址？该条件转移指令向后（反向）最多可跳转多少条指令？

（2）某条件转移指令的地址为 200CH，指令内容如下图所示，若该指令执行时 CF=0，ZF=0，NF=1，则该指令执行后 PC 的值是多少？若该指令执行时 CF=1，ZF=0，NF=0，则该指令执行后 PC 的值又是多少？请给出计算过程。

15　　11	10	9	8	7　　0
00000	0	1	1	11100011

（3）实现“无符号数比较小于等于时转移”功能的指令中，C、Z 和 N 应各是什么？

（4）以下是该指令对应的数据通路示意图，要求给出图中部件①～③的名称或功能说明。



（1）因为指令长度为 16 位，且下条指令地址为(PC)+2，故编址单位是字节。

偏移量 OFFSET 为 8 位补码，范围为−128～127，故相对于当前条件转移指令，向后最多可跳转 127 条指令。

（2）指令中 C=0，Z=1，N=1，故应根据 ZF 和 NF 的值来判断是否转移。当 CF=0，ZF=0，NF=1 时，需转移。已知指令中偏移量为 1110 0011B=E3H，符号扩展后为 FFE3H，左移一位（乘 2）后为 FFC6H，故 PC 的值（即转移目标地址）为 200CH+2+FFC6H=1FD4H。当 CF=1，ZF=0，NF=0 时不转移。PC 的值为：200CH+2=200EH。

（3）指令中的 C、Z 和 N 应分别设置为 C=Z=1，N=0。

（4）部件①：指令寄存器（用于存放当前指令）；部件②：移位寄存器（用于左移一位）；部件③：加法器（地址相加）。

数据总线的位数与 CPU 的位数相同，表示了 CPU 一次能处理的数据的位数，即多少位的 CPU

指令寄存器存放当前执行的指令，对用户是透明的，而 CPU 的状态寄存器是汇编程序员可见的

CPU 从内存中取出一条指令并执行完毕的时间叫做指令周期，计算机的最小时间单位是时钟周期，CPU 时钟周期就是时钟周期。CPU 周期就是机器周期，它包含 n 个时钟周期。时钟周期≤指令周期≤机器周期。采用 DMA 方式时，每传输一个数据块就需要一个存取周期。指令周期的第一个机器周期是取出指令的时间，叫做取指周期，如果指令字长是存储字长的 n 倍，就需要 n 次访问内存，取指周期就是机器周期的 n 倍。因为 CPU 和内存的巨大的速度差异，机器周期通常由存取周期来确定

在单总线的 CPU 中 ALU 只能有一个输入端与总线连接，另外的输入端要通过寄存器与总线连接

CPU 或者任何的计算机器都可以分为两部分：CU（control，控制器）和 PU（process，执行，或者叫运算器），目前 CPU 的控制部件有两类：微程序、组合逻辑（硬布线）

在微程序控制器中，微命令：控制器发出的控制信号，微操作：微命令执行的操作，微指令：若干微命令的集合，微程序：若干微命令的有序集合。一个微程序的周期对应一个机器周期。一个微程序的作用就是要实现一条指令。控制信号以微程序的形式专门存放在一个存储器（控制存储器，在 CPU 内部，由 ROM 组成）当中，执行指令的时候读出来即可。如果计算机有 n 种指令，那么这个存储器里面的微程序至少是 n+1（还有 1 个公共的取指微程序，如果系统有中断功能，就还要 1 个对应中断周期的微程序）

注意微程序控制器不是微处理器，微处理器的意思就是一个小小的 CPU

取出来的指令的操作码会告诉你微程序的入口地址

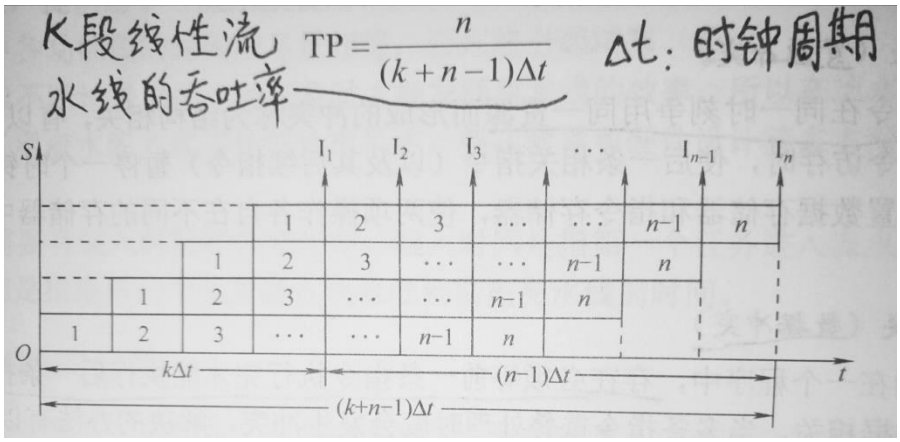
	微程序	组合逻辑
控制信号的来源	从专门的控制存储器中取出	电路当前的即时状态
速度	慢	快
特点	规范，易修改	繁琐，难修改

流水线是一种把重复的过程分解为若干子过程，每个子过程和其他子过程并行执行的技术

吞吐率：单位时间内流水线完成的任务数目

m 段流水线稳定时 CPU 的吞吐率等于具有 m 个并行部件的 CPU 的吞吐率

流水线的 3 种冒险（hazard）：结构冒险（多条指令同时抢同一个资源，解决方法：增加硬件资源、暂停流水线）、数据冒险（后一条指令要用前一条指令的结果，解决方法：使用专门的数据旁路技术、暂停流水线）、控制冒险（遇到转移指令造成断流，解决方法：编译器优化）



并不是流水线的并行的段越多，效率就越高。对流水线危害最大的冒险是数据冒险

总线宽度：总线一次能传递的数据的位数，即数据总线的条数

总线的时钟周期＝1/总线的时钟频率

总线带宽（最大数据传输率）=总线的时钟频率×总线宽度/8　　单位：字节/秒

连接到总线的部件和总线的数据传输的特点是：分时发送，同时接收

总线的突发传输：从设备从主设备给出的首地址的开始的若干连续单元读写多个数据

44. （12 分）某程序中有如下循环代码段 p “for(int i = 0; i < N; i++) sum+=A[i];”。假设编译时变量 sum 和 i 分别分配在寄存器 R1 和 R2 中。常量 N 在寄存器 R6 中，数组 A 的首地址在寄存器 R3 中。程序段 P 起始地址为 0804 8100H，对应的汇编代码和机器代码如下表所示。

编号	地址	机器代码	汇编代码	注释
1	08048100H	00022080H	loop: sll R4,R2,2	(R2)<<2 → R4
2	08048104H	00083020H	add R4,R4,R3	(R4)+(R3) → R4
3	08048108H	8C850000H	load R5,0(R4)	((R4)+0) → R5
4	0804810CH	00250820H	add R1,R1,R5	(R1)+(R5) → R1
5	08048110H	20420001H	add R2,R2,1	(R2)+1 → R2
6	08048114H	1446FFFAH	bne R2,R6,loop	if(R2)!=(R6) goto loop

执行上述代码的计算机 M 采用 32 位定长指令字，其中分支指令 bne 采用如下格式：

31　　26	25　　21	20　　16	15　　0
OP	Rs	Rd	OFFSET

OP 为操作码；Rs 和 Rd 为寄存器编号；OFFSET 为偏移量，用补码表示。请回答下列问题，并说明理由。

1）M 的存储器编址单位是什么？

2）已知 sll 指令实现左移功能，数组 A 中每个元素占多少位？

3）题 44 表中 bne 指令的 OFFSET 字段的值是多少？已知 bne 指令采用相对寻址方式，当前 PC 内容为 bne 指令地址，通过分析题 44 表中指令地址和 bne 指令内容，推断出 bne 指令的转移目标地址计算公式。

4）若 M 采用如下“按序发射、按序完成”的 5 级指令流水线：IF（取值）、ID（译码及取数）、EXE（执行）、MEM（访存）、WB（写回寄存器），且硬件不采取任何转发措施，分支指令的执行均引起 3 个时钟周期的阻塞，则 P 中哪些指令的执行会由于数据相关而发生流水线阻塞？哪条指令的执行会发生控制冒险？为什么指令 1 的执行不会因为与指令 5 的数据相关而发生阻塞？

45. 假设对于 44 题中的计算机 M 和程序 P 的机器代码，M 采用页式虚拟存储管理；P 开始执行时，(R1)=(R2)=0，(R6)=1000，其机器代码已调入主存但不在 Cache 中；数组 A 未调入主存，且所有数组元素在同一页，并存储在磁盘同一个扇区。请回答下列问题并说明理由。

1）P 执行结束时，R2 的内容是多少？

2）M 的指令 Cache 和数据 Cache 分离。若指令 Cache 共有 16 行，Cache 和主存交换的块大小为 32 字节，则其数据区的容量是多少？若仅考虑程序段 P 的执行，则指令 Cache 的命中率为多少？

3）P 在执行过程中，哪条指令的执行可能发生溢出异常？哪条指令的执行可能产生缺页异常？对于数组 A 的访问，需要读磁盘和 TLB 至少各多少次？

（来源：2014 年硕士研究生入学统考计算机专业课 408 真题）

44．解答：

1）已知计算机 M 采用 32 位定长指令字，即一条指令占 4B，观察表中各指令的地址可知，每条指令的地址差为 4 个地址单位，即 4 个地址单位代表 4B，一个地址单位就代表了 1B，所以该计算机是按字节编址的。

2）在二进制中某数左移二位相当于乘以四，由该条件可知，数组间的数据间隔为 4 个地址单位，而计算机按字节编地址，所以数组 A 中每个元素占 4B。

3）由表可知，bne 指令的机器代码为 1446FFFAH，根据题目给出的指令格式，后 2B 的内容为 OFFSET 字段，所以该指令的 OFFSET 字段为 FFFAH，用补码表示，值为-6。当系统执行到 bne 指令时，PC 自动加 4，PC 的内容就为 08048118H，而跳转的目标是 08048100H，两者相差了 18H，即 24 个单位的地址间隔，所以偏移地址的一位即是真实跳转地址的-24/-6=4 位。可知 bne 指令的转移目标地址计算公式为(PC)+4+OFFSET*4。

4）由于数据相关而发生阻塞的指令为第 2、3、4、6 条，因为第 2、3、4、6 条指令都与各自前一条指令发生数据相关。

第 6 条指令会发生控制冒险。

当前循环的第五条指令与下次循环的第一条指令虽然有数据相关，但由于第 6 条指令后有 3 个时钟周期的阻塞，因而消除了该数据相关。

45．解答：

1）R2 里装的是 i 的值，循环条件是 i<N(1000)，即当 i 自增到不满足这个条件时跳出循环，程序结束，所以此时 i 的值为 1000。

2）Cache 共有 16 块，每块 32 字节，所以 Cache 数据区的容量为 16*32B=512B。

P 共有 6 条指令，占 24 字节，小于主存块大小(32B)，其起始地址为 0804 8100H，对应一块的开始位置，由此可知所有指令都在一个主存块内。读取第一条指令时会发生 Cache 缺失，故将 P 所在的主存块调入 Cache 某一块，以后每次读取指令时，都能在指令 Cache 中命中。因此在 1000 次循环中，只会发生 1 次指令访问缺失，所以指令 Cache 的命中率为：(1000×6-1)/(1000×6)=99.98%。

3）指令 4 为加法指令，即对应 sum+=A[i]，当数组 A 中元素的值过大时，则会导致这条加法指令发生溢出异常；而指令 2、5 虽然都是加法指令，但他们分别为数组地址的计算指令和存储变量 i 的寄存器进行自增的指令，而 i 最大到达 1000，所以他们都不会产生溢出异常。

只有访存指令可能产生缺页异常，即指令 3 可能产生缺页异常。

因为数组 A 在磁盘的一页上，而一开始数组并不在主存中，第一次访问数组时会导致访盘，把 A 调入内存，而以后数组 A 的元素都在内存中，则不会导致访盘，所以该程序一共访盘一次。

每访问一次内存数据就会查 TLB 一次，共访问数组 1000 次，所以此时又访问 TLB1000 次，还要考虑到第一次访问数组 A，即访问 A[0]时，会多访问一次 TLB（第一次访问 A[0]会先查一次 TLB，然后产生缺页，处理完缺页中断后，会重新访问 A[0]，此时又查 TLB），所以访问 TLB 的次数一共是 1001 次。

动态性是进程的最基本特征，是进程和程序的最基本区别

进程控制块（PCB）是进程存在的唯一标志，进程创建之后进入就绪队列，它的 PCB 常驻内存，进程结束时删除。如果同一个程序运行了两次，算作两个不同的进程

进程是分配资源的基本单位，线程是 CPU 调度的基本单位

线程的实现又可分为用户级（内核意识不到线程的存在，线程管理由用户程序完成）和内核级。

同一进程或不同进程内的线程都可以并发执行，线程的切换可能引起进程的切换

进程创建的若干个线程可以共享进程内的资源，比如进程中打开的文件

管道是用于实现进程之间通信的共享文件，进程对管道进行读/写都可能被阻塞

C 语言程序在使用内存时分为 3 个段：正文段（全局变量、常量）、堆段（分配的动态内存）、

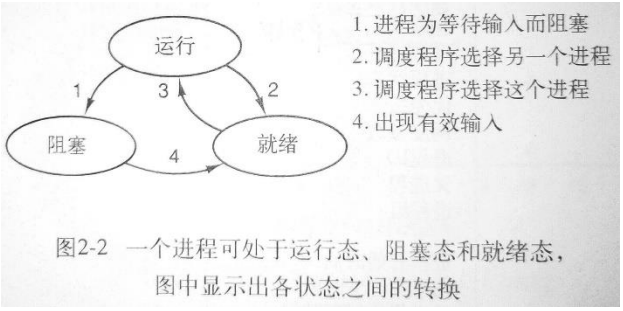


图2-2 一个进程可处于运行态、阻塞态和就绪态，图中显示出各状态之间的转换

栈段（未赋值的局部变量、实参传递）

并发进程失去封闭性是指并发进程共享变量，其执行结果与速度有关

键盘输入是人的操作，是远远慢于计算机的，所以完全可以用一个线程处理整个系统的键盘输入

全局变量是对同一进程而言的，在不同的进程中是不同的变量

操作系统用原语来管理和控制进程。原子操作是指一组相关联的操作要么都不间断的执行，要么都不执行

进程申请读磁盘操作的时候，因为要等待 I/O 操作的完成，所以会把自己阻塞，I/O 操作完毕以后，进程得到了想要的资源就会转为就绪态。进程从阻塞→就绪称为唤醒这个进程

周转时间：作业从提交到完成　　响应时间：用户提交请求到系统首次产生响应

响应比：响应时间÷要求服务时间＝（等待时间+要求服务时间）÷要求服务时间

进程调度策略：

	先来先服务	短作业优先	高响应比优先	时间片轮转	多级反馈队列
抢占性	必须是不可抢	不一定		必须是可以抢	队列内的算法不一定
优点	公平简单	平均等待时间最短	兼顾长短作业		
缺点	不利于短作业、I/O 忙的作业	长作业会饥饿	计算响应比的开销大	等待时间长，上下文切换浪费时间	
适用于		批处理系统		分时系统	
抢占性默认	非抢占			抢占	

一次仅允许一个进程使用的资源叫做临界资源，进程中访问临界资源的那部分代码叫做临界区

信号量用于解决互斥和同步的问题，同时引入两个原子操作：

P：若信号量＞0，则信号量的值减 1　　若信号量=0，则进程会睡眠（就绪→阻塞）

V：信号量的值加 1

信号量是一个特殊的整型变量，只有初始化和 PV 操作能改变它的值，初值由用户来确定。信号量通常分为互斥量和资源量，互斥量的初值一般为 1，表示临界区只允许 1 个进程进入，从而实现互斥，当互斥量为 0 时，表示临界区已经有一个进程进入，临界区外没有进程等待；当互斥量小于 0 时，表示临界区已经有一个进程进入，互斥量的绝对值表示临界区外等待的进程数目。资源量为 0 时表示所有资源已经用完，资源量大于 0 时表示还可以用的资源数量，资源量小于 0 时，资源量的绝对值表示正在等待该资源的进程数目。

如果一个进程集合中的每个进程都在等待只能由该进程集合中的其他进程才能引发的事件，那么这个进程集合就是死锁的。

必须同时满足以下四个条件，才会发生死锁：

45.（9 分）有 A、B 两人通过信箱进行辩论，每个人都从自己的信箱中取得对方的问题。将答案和向对方提出的新问题组成一个邮件放入对方的邮箱中。假设 A 的信箱最多放 M 个邮件，B 的信箱最多放 N 个邮件。初始时 A 的信箱中有 x 个邮件（0<x<M），B 的信箱中有 y 个（0<y<N）。辩论者每取出一个邮件，邮件数减 1。A 和 B 两人的操作过程描述如下：

CoBegin

A{
while(TRUE){
从 A 的信箱中取出一个邮件；
回答问题并提出一个新问题；
将新邮件放入 B 的信箱；
}
}
	B{
	while(TRUE){
	从 B 的信箱中取出一个邮件；
	回答问题并提出一个新问题；
	将新邮件放入 A 的信箱；
	}
	}

CoEnd

当信箱不为空时，辩论者才能从信箱中取邮件，否则等待。当信箱不满时，辩论者才能将新邮件放入信箱，否则等待。请添加必要的信号量和 P、V（或 wait、signal）操作，以实现上述过程的同步。要求写出完整过程，并说明信号量的含义和初值。

```
semaphore Full_A = x;          //Full_A 表示 A 的信箱中的邮件数量
semaphore Empty_A = M-x;       // Empty_A 表示 A 的信箱中还可存放的邮件数量
semaphore Full_B = y;          //Full_B 表示 B 的信箱中的邮件数量
semaphore Empty_B = N-y;       // Empty_B 表示 B 的信箱中还可存放的邮件数量
semaphore mutex_A = 1;         //mutex_A 用于 A 的信箱互斥
semaphore mutex_B = 1;         //mutex_B 用于 B 的信箱互斥
Cobegin
```

A{
while(TRUE){
P(Full_A);
P(mutex_A);
从 A 的信箱中取出一个邮件;
V(mutex_A);
V(Empty_A);
回答问题并提出一个新问题;
P(Empty_B);
P(mutex_B);
将新邮件放入 B 的信箱;
	B{
	while(TRUE){
	P(Full_B);
	P(mutex_B);
	从 B 的信箱中取出一个邮件;
	V(mutex_B);
	V(Empty_B);
	回答问题并提出一个新问题;
	P(Empty_A);
	P(mutex_A);
	将新邮件放入 A 的信箱;

V(mutex_B); V(Full_B); } }	V(mutex_A); V(Full_A); } }

互斥条件：资源要么是可用的，要么只被一个进程占用

不可抢占条件：只能由占有了资源的这个进程来主动释放资源，其他进程不能抢占

占有和等待条件：已经占有某个资源的进程可以再请求其他资源

环路等待条件：存在一个进程环路，环路中每个进程都在等待下一个进程占有的资源

死锁预防：破坏四个条件之一，限制用户申请资源的顺序
死锁避免：防止系统进入不安全状态，银行家算法。不会对资源分配过程作任何干扰
死锁检测：资源分配图，死锁定理。不会对资源分配过程作任何干扰
死锁解除：抢占资源，撤销进程
死锁状态是不安全状态的子集
饥饿：进程在信号量内无限期等待。处于饥饿状态的进程可以是一个就绪的进程，处于死锁状态的进程一定是一个阻塞的进程
某系统有 8 台打印机，由 K 个进程竞争使用，每个进程最多需要 3 台打印机，该系统可能会发生死锁的 K 的最小值是？
根据抽屉原理，如果每个进程都已经占了 2 台打印机，那么只要还有多的打印机，就总能满足某个进程占了 3 台打印机的情况，然后顺利执行下去。所以 K 的最小值是 8/2=4
经典的进程同步问题有读者/写者问题、生产者/消费者问题、哲学家就餐问题
实际上在 Linux 操作系统中，线程之间同步才需要互斥量、读写锁，进程之间通信需要信号量、匿名管道、TCP socket、共享内存、文件裸奔（文件的存在与否来当做锁）、消息队列，进程主要是被调度的。能通信就一定能作为一种同步机制，不要把"同步"和"通信"分得太清楚（读书读傻了）

C++多态性与虚函数表

C++多态的实现？

多态分为静态多态和动态多态。静态多态是通过重载和模板技术实现，在编译的时候确定。动态多态通过虚函数和继承关系来实现，执行动态绑定，在运行的时候确定。

动态多态实现有几个条件：

(1) 虚函数；

(2) 一个基类的指针或引用指向派生类的对象；

基类指针在调用成员函数(虚函数)时，就会去查找该对象的虚函数表。虚函数表的地址在每个对象的首地址。查找该虚函数表中该函数的指针进行调用。

每个对象中保存的只是一个虚函数表的指针，C++内部为每一个类维持一个虚函数表，该类的对象的都指向这同一个虚函数表。

虚函数表中为什么就能准确查找相应的函数指针呢？因为在类设计的时候，虚函数表直接从基类也继承过来，如果覆盖了其中的某个虚函数，那么虚函数表的指针就会被替换，因此可以根据指针准确找到该调用哪个函数。

虚函数的作用？

虚函数用于实现多态，这点大家都能答上来

但是虚函数在设计上还具有封装和抽象的作用。比如抽象工厂模式。

动态绑定是如何实现的？

第一个问题中基本回答了，主要都是结合虚函数表来答就行。

静态多态和动态多态。静态多态是指通过模板技术或者函数重载技术实现的多态，其在编译器确定行为。动态多态是指通过虚函数技术实现在运行期动态绑定的技术。

虚函数表

虚函数表是针对类的还是针对对象的？同一个类的两个对象的虚函数表是怎么维护的？

编译器为每一个类维护一个虚函数表，每个对象的首地址保存着该虚函数表的指针，同一个类的不同对象实际上指向同一张虚函数表。

纯虚函数如何定义，为什么对于存在虚函数的类中析构函数要定义成虚函数

为了实现多态进行动态绑定，将派生类对象指针绑定到基类指针上，对象销毁时，如果析构函数没有定义为析构函数，则会调用基类的析构函数，显然只能销毁部分数据。如果要调用对象的析构函数，就需要将该对象的析构函数定义为虚函数，销毁时通过虚函数表找到对应的析构函数。

2//纯虚函数定义

virtual ~myClass() = 0;

析构函数能抛出异常吗

答案肯定是不能。 C++标准指明析构函数不能、也不应该抛出异常。C++异常处理模型最大的特点和优势就是对 C++中的面向对象提供了最强大的无缝支持。那么如果对象在运行期间出现了异常，C++异常处理模型有责任清除那些由于出现异常所导致的已经失效了的对象(也即对象超出了它原来的作用域)，并释放对象原来所分配的资源， 这就是调用这些对象的析构函数来完成释放资源的任务，所以从这个意义上说，析构函数已经变成了异常处理的一部分。

(1) 如果析构函数抛出异常，则异常点之后的程序不会执行，如果析构函数在异常点之后执行了某些必要的动作比如释放某些资源，则这些动作不会执行，会造成诸如资源泄漏的问题。

(2) 通常异常发生时，c++的机制会调用已经构造对象的析构函数来释放资源，此时若析构函数本身也抛出异常，则前一个异常尚未处理，又有新的异常，会造成程序崩溃的问题。

构造函数和析构函数中调用虚函数吗？

指针和引用的区别

指针保存的是所指对象的地址，引用是所指对象的别名，指针需要通过解引用间接访问，而引用是直接访问；

指针可以改变地址，从而改变所指的对象，而引用必须从一而终；

引用在定义的时候必须初始化，而指针则不需要；

指针有指向常量的指针和指针常量，而引用没有常量引用；

指针更灵活，用的好威力无比，用的不好处处是坑，而引用用起来则安全多了，但是比较死板。

指针与数组千丝万缕的联系

一个一维 int 数组的数组名实际上是一个 int* const 类型；

一个二维 int 数组的数组名实际上是一个 int (*const p)[n];

数组名做参数会退化为指针，除了 sizeof

智能指针是怎么实现的？什么时候改变引用计数？

构造函数中计数初始化为 1；

拷贝构造函数中计数值加 1；

赋值运算符中，左边的对象引用计数减一，右边的对象引用计数加一；

析构函数中引用计数减一；

在赋值运算符和析构函数中，如果减一后为 0，则调用 delete 释放对象

十亿整数（随机生成，可重复）中前 K 最大的数

类似问题的解决方法思路：首先哈希将数据分成 N 个文件，然后对每个文件建立 K 个元素最小/大堆（根据要求来选择）。最后将文件中剩余的数插入堆中，并维持 K 个元素的堆。最后将 N 个堆中的元素合起来分析。可以采用归并的方式来合并。在归并的时候为了提高效率还需要建一个 N 个元素构成的最大堆，先用 N 个堆中的最大值填充这个堆，然后就是弹出最大值，指针后移的操作了。当然这种问题在现在的互联网技术中，一般就用 map-reduce 框架来做了。

大数据排序相同的思路：先哈希（哈希是好处是分布均匀，相同的数在同一个文件中），然后小文件装入内存快排，排序结果输出到文件。最后建堆归并

get/post 区别

区别一：

get 重点在从服务器上获取资源，post 重点在向服务器发送数据；

区别二：

get 传输数据是通过 URL 请求，以 field（字段）= value 的形式，置于 URL 后，并用"?"连接，多个请求数据间用"&"连接，如 http://127.0.0.1/Test/login.action?name=admin&password=admin，这个过程用户是可见的；

post 传输数据通过 Http 的 post 机制，将字段与对应值封存在请求实体中发送给服务器，这个过程对用户是不可见的；

区别三：

Get 传输的数据量小，因为受 URL 长度限制，但效率较高；

Post 可以传输大量数据，所以上传文件时只能用 Post 方式；

区别四：

get 是不安全的，因为 URL 是可见的，可能会泄露私密信息，如密码等；

post 较 get 安全性较高；

浏览器中输入一个 URL 发生什么，用到哪些协议？

浏览器中输入 URL，首先浏览器要将 URL 解析为 IP 地址，解析域名就要用到 DNS 协议，首先主机会查询 DNS 的缓存，如果没有就给本地 DNS 发送查询请求。DNS 查询分为两种方式，一种是递归查询，一种是迭代查询。如果是迭代查询，本地的 DNS 服务器，向根域名服务器发送查询请求，根域名服务器告知该域名的一级域名服务器，然后本地服务器给该一级域名服务器发送查询请求，然后依次类推直到查询到该域名的 IP 地址。DNS 服务器是基于 UDP 的，因此会用到 UDP 协议。

得到 IP 地址后，浏览器就要与服务器建立一个 http 连接。因此要用到 http 协议，http 协议报文格式上面已经提到。http 生成一个 get 请求报文，将该报文传给 TCP 层处理。如果采用 https 还会先对 http 数据进行加密。TCP 层如果有需要先将 HTTP 数据包分片，分片依据路径 MTU 和 MSS。TCP 的数据包然后会发送给 IP 层，用到 IP 协议。IP 层通过路由选路，一跳一跳发送到目的地址。当然在一个网段内的寻址是通过以太网协议实现(也可以是其他

物理层协议，比如 PPP，SLIP)，以太网协议需要直到目的 IP 地址的物理地址，有需要 ARP 协议。

Linux 中 select，poll，epoll 的区别

select :是最初解决 IO 阻塞问题的方法。用结构体 fd_set 来告诉内核监听多个文件描述符，该结构体被称为描述符集。由数组来维持哪些描述符被置位了。对结构体的操作封装在三个宏定义中。通过轮寻来查找是否有描述符要被处理，如果没有返回

存在的问题：

1. 内置数组的形式使得 select 的最大文件数受限与 FD_SIZE；
2. 每次调用 select 前都要重新初始化描述符集，将 fd 从用户态拷贝到内核态，每次调用 select 后，都需要将 fd 从内核态拷贝到用户态；
3. 轮寻排查当文件描述符个数很多时，效率很低；

poll 通过一个可变长度的数组解决了 select 文件描述符受限的问题。数组中元素是结构体，该结构体保存描述符的信息，每增加一个文件描述符就向数组中加入一个结构体，结构体只需要拷贝一次到内核态。poll 解决了 select 重复初始化的问题。轮寻排查的问题未解决。

epoll：轮寻排查所有文件描述符的效率不高，使服务器并发能力受限。因此，epoll 采用只返回状态发生变化的文件描述符，便解决了轮寻的瓶颈。

- 为什么使用 IO 多路复用，最主要的原因是什么？
- epoll 有两种触发模式？这两种触发模式有什么区别？编程的时候有什么区别？
- 上一题中编程的时候有什么区别，是在边缘触发的时候要把套接字中的数据读干净，那么当有多个套接字时，在读的套接字一直不停的有数据到达，如何保证其他套接字不被饿死(面试网易游戏的时候问的一个问题，答不上来，印象贼深刻)

常见的数据库面试题有哪些呢？（非 DBA 向）

（一）什么是存储过程？有哪些优缺点？

存储过程是一些预编译的 SQL 语句。

更加直白的理解：存储过程可以说是一个记录集，它是由一些 T-SQL 语句组成的代码块，这些 T-SQL 语句代码像一个方法一样实现一些功能（对单表或多表的增删改查），然后再给这个代码块取一个名字，在用到这个功能的时候调用他就行了。

存储过程是一个预编译的代码块，执行效率比较高

一个存储过程替代大量 T_SQL 语句 ，可以降低网络通信量，提高通信速率

可以一定程度上确保数据安全

（二）索引是什么？有什么作用以及优缺点？

索引是对数据库表中一或多个列的值进行排序的结构，是帮助 MySQL 高效获取数据的数据结构

你也可以这样理解：索引就是加快检索表中数据的方法。数据库的索引类似于书籍的索引。在书籍中，索引允许用户不必翻阅完整个书就能迅速地找到所需要的信息。在数据库中，索引也允许数据库程序迅速地找到表中的数据，而不必扫描整个数据库。

MySQL 数据库几个基本的索引类型：普通索引、唯一索引、主键索引、全文索引

索引加快数据库的检索速度

索引降低了插入、删除、修改等维护任务的速度

唯一索引可以确保每一行数据的唯一性

通过使用索引，可以在查询的过程中使用优化隐藏器，提高系统的性能

索引需要占物理和数据空间

（三）什么是事务？

事务（Transaction）是并发控制的基本单位。所谓的事务，它是一个操作序列，这些操作要么都执行，要么都不执行，它是一个不可分割的工作单位。事务是数据库维护数据一致性的单位，在每个事务结束时，都能保持数据一致性。

（四）数据库的乐观锁和悲观锁是什么？

数据库管理系统（DBMS）中的并发控制的任务是确保在多个事务同时存取数据库中同一数据时不破坏事务的隔离性和统一性以及数据库的统一性。

乐观并发控制(乐观锁)和悲观并发控制（悲观锁）是并发控制主要采用的技术手段。

悲观锁：假定会发生并发冲突，屏蔽一切可能违反数据完整性的操作

乐观锁：假设不会发生并发冲突，只在提交操作时检查是否违反数据完整性。

（五） 使用索引查询一定能提高查询的性能吗？为什么

通常,通过索引查询数据比全表扫描要快.但是我们也必须注意到它的代价.

索引需要空间来存储,也需要定期维护, 每当有记录在表中增减或索引列被修改时,索引本身也会被修改. 这意味着每条记录的 INSERT,DELETE,UPDATE 将为此多付出 4,5 次的磁盘 I/O. 因为索引需要额外的存储空间和处理,那些不必要的索引反而会使查询反应时间变慢.使用索引查询不一定能提高查询性能,索引范围查询(INDEX RANGE SCAN)适用于两种情况:

基于一个范围的检索,一般查询返回结果集小于表中记录数的 30%

基于非唯一性索引的检索

（六）简单说一说 drop、delete 与 truncate 的区别

SQL 中的 drop、delete、truncate 都表示删除，但是三者有一些差别

delete 和 truncate 只删除表的数据不删除表的结构

速度,一般来说: drop> truncate >delete

delete 语句是 dml,这个操作会放到 rollback segement 中,事务提交之后才生效;

如果有相应的 trigger,执行的时候将被触发. truncate,drop 是 ddl, 操作立即生效,原数据不放到 rollback segment 中,不能回滚. 操作不触发 trigger.

（七）drop、delete 与 truncate 分别在什么场景之下使用？

不再需要一张表的时候，用 drop

想删除部分数据行时候，用 delete，并且带上 where 子句

保留表而删除所有数据的时候用 truncate

（八） 超键、候选键、主键、外键分别是什么？

超键：在关系中能唯一标识元组的属性集称为关系模式的超键。一个属性可以为作为一个超键，多个属性组合在一起也可以作为一个超键。超键包含候选键和主键。

候选键：是最小超键，即没有冗余元素的超键。

主键：数据库表中对储存数据对象予以唯一和完整标识的数据列或属性的组合。一个数据列只能有一个主键，且主键的取值不能缺失，即不能为空值（Null）。

外键：在一个表中存在的另一个表的主键称此表的外键。

（九）什么是视图？以及视图的使用场景有哪些？

视图是一种虚拟的表，具有和物理表相同的功能。可以对视图进行增，改，查，操作，试图通常是有一个表或者多个表的行或列的子集。对视图的修改不影响基本表。它使得我们获取数据更容易，相比多表查询。

只暴露部分字段给访问者，所以就建一个虚表，就是视图。

查询的数据来源于不同的表，而查询者希望以统一的方式查询，这样也可以建立一个视图，把多个表查询结果联合起来，查询者只需要直接从视图中获取数据，不必考虑数据来源于不同表所带来的差异

（十）说一说三个范式。

第一范式（1NF）：数据库表中的字段都是单一属性的，不可再分。这个单一属性由基本类型构成，包括整型、实数、字符型、逻辑型、日期型等。

第二范式（2NF）：数据库表中不存在非关键字段对任一候选关键字段的部分函数依赖（部分函数依赖指的是存在组合关键字中的某些字段决定非关键字段的情况），也即所有非关键字段都完全依赖于任意一组候选关键字。

第三范式（3NF）：在第二范式的基础上，数据表中如果不存在非关键字段对任一候选关键字段的传递函数依赖则符合第三范式。所谓传递函数依赖，指的是如 果存在"A → B → C"的决定关系，则 C 传递函数依赖于 A。因此，满足第三范式的数据库表应该不存在如下依赖关系： 关键字段 → 非关键字段 x → 非关键字段 y

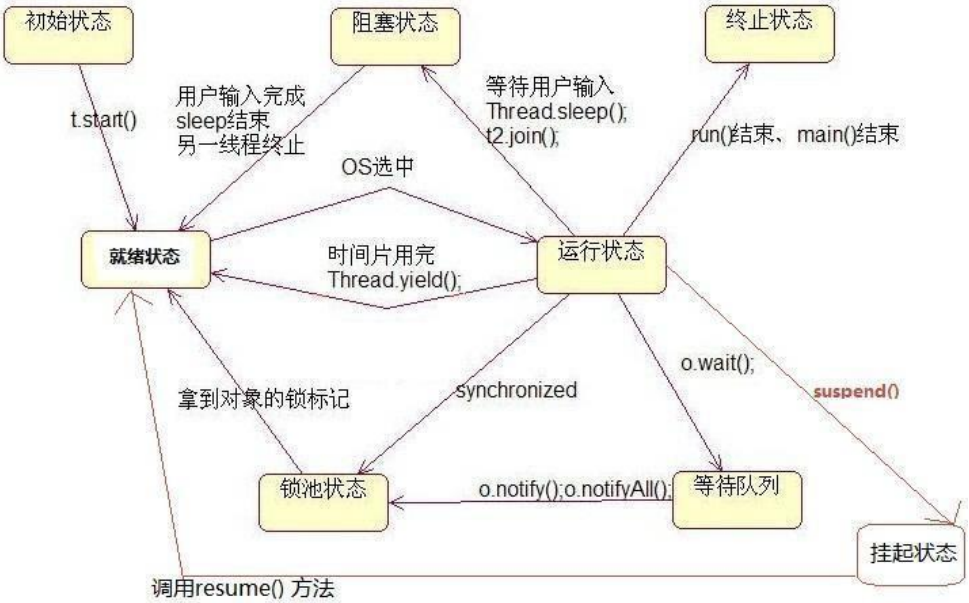
---

C/C++ 的 `static` 有 3 个（不太相关的）用途：

1. 内部链接（C/C++）的全局函数 / 变量声明使用`static`修饰，表示该函数 / 变量只在所在的编译单元内可见。与之相关的是外部链接声明，使用`extern`修饰。
2. 静态局部变量（C/C++）在程序第一次运行到该声明时做初始化，作用域结束后变量不被消毁，下一次运行时仍保持状态。普通的局部变量称为自动变量。
3. 静态成员函数 / 变量（C++）表示函数 / 变量不绑定至实例。静态变量除了声明外还需要在类之外显式定义。

```
static int a = 0;      // 1. 内部链接的变量

static void f() {      // 1. 内部链接的函数
}

int g() {
    static int b = 0; // 2. 静态局部变量
    b++;
    return b;
}

class Foo {
    static void h();  // 3. 静态成员函数声明
    static int c;     // 3. 静态成员变量声明
};

void Foo::h() {       // 3. 静态成员函数定义
}

int Foo::c = 0;       // 3. 静态成员变量定义
```

Java 中线程的状态转换？



catch 中的 return 在 finally 之后执行 但是会将 return 的值放到一个地方存起来

1.SQL 指的是？

您的回答：Structured Query Language

2.哪个 SQL 语句用于从数据库中提取数据？

您的回答：SELECT

3.哪条 SQL 语句用于更新数据库中的数据？

您的回答：UPDATE

4.哪条 SQL 语句用于删除数据库中的数据？

您的回答：DELETE

5.哪条 SQL 语句用于在数据库中插入新的数据？

您的回答：INSERT INTO

6.通过 SQL，您如何从 "Persons" 表中选取 "FirstName" 列

您的回答：SELECT FirstName FROM Persons

7.通过 SQL，您如何从 "Persons" 表中选取所有的列？

您的回答：SELECT * FROM Persons

8.通过 SQL，您如何从 "Persons" 表中选取 "FirstName" 列的值等于"Peter" 的所有记录？

您的回答：SELECT * FROM Persons WHERE FirstName='Peter'

9.通过 SQL，您如何从 "Persons" 表中选取 "FirstName" 列的值以 "a" 开头的所有记录？

您的回答：SELECT * FROM Persons WHERE FirstName LIKE 'a%'

10.请判断下列说法是否正确：当所列出的某个条件为 true 时，OR 运算符会显示记录。当列出的所有条件为 true 时，AND 运算符会显示记录。

您的回答：正确

11.通过 SQL，您如何在表 Persons 中选择 FirstName 等于 Thomas 而 LastName 等于 Carter 的所有记录？

您的回答：SELECT * FROM Persons WHERE FirstName='Thomas' AND LastName='Carter'

12.通过 SQL，您如何按字母顺序选取 Persons 表中 LastName 介于 Adams 和 Carter 的所有记录？

您的回答：SELECT * FROM Persons WHERE LastName BETWEEN 'Adams' AND 'Carter'

13.哪条 SQL 语句可返回唯一不同的值？

您的回答：SELECT DISTINCT

14.哪个 SQL 关键词用于对结果集进行排序？

您的回答：ORDER BY

15.通过 SQL，您如何根据 "FirstName" 列按序地从 "Persons" 表返回所有记录？

您的回答：SELECT * FROM Persons ORDER BY FirstName DESC

16.通过 SQL，您如何向 "Persons" 表插入新的记录？

您的回答：INSERT INTO Persons VALUES ('Jimmy', 'Jackson')

17.通过 SQL，您如何向 "Persons" 表中的 "LastName" 列插入 "Wilson" ？

您的回答：INSERT INTO Persons (LastName) VALUES ('Wilson')

18.您如何把 "Persons" 表中 "LastName" 列的 "Gates" 改为 "Wilson" ？

您的回答：UPDATE Persons SET LastName='Wilson' INTO LastName='Gates'

正确答案：UPDATE Persons SET LastName='Wilson' WHERE LastName='Gates'

19.通过 SQL，您如何在 "Persons" 表中删除 "FirstName" 等于 "Fred" 的纪录？

您的回答：DELETE FROM Persons WHERE FirstName = 'Fred'

20.通过 SQL，您如何返回 "Persons" 表中记录的数目？

您的回答：SELECT COUNT(*) FROM Persons

24.优化问题：组团旅游.py

为来自不同地方而要去同一地点的人(假设这是一家人)安排一次旅游，家庭成员来自全国各地，它们希望在纽约的 LGA 机场会面，他们想在同一天到达并在同一天离开。每天都有许多航班从任何一个人的所在地飞往纽约，这些飞机的起飞时间和价格、续航时间都不尽相同，为了了解这些航班信息，去网站

https://resources.oreilly.com/examples/9780596529321/blob/master/PCI_Code%20Folder/chapter5/schedule.txt

上找到了这些数据：

```
LGA,OMA,6:19,8:13,239     LGA,ORD,14:19,17:09,190  LGA,BOS,6:39,8:09,86      LGA,DAL,14:20,17:32,332
OMA,LGA,6:11,8:31,249     ORD,LGA,14:22,16:32,126  BOS,LGA,6:17,8:26,89      DAL,LGA,13:54,18:02,294
LGA,OMA,8:04,10:59,136    LGA,ORD,15:04,17:23,189  LGA,BOS,8:23,10:28,149    LGA,DAL,15:49,20:10,497
OMA,LGA,7:39,10:24,219    ORD,LGA,15:58,18:40,173  BOS,LGA,8:04,10:11,95     DAL,LGA,15:44,18:55,382
LGA,OMA,9:31,11:43,210    LGA,ORD,17:06,20:00,95   LGA,BOS,9:58,11:18,130    LGA,DAL,17:14,20:59,277
OMA,LGA,9:15,12:03,99     ORD,LGA,16:43,19:00,246  BOS,LGA,9:45,11:50,172    DAL,LGA,16:52,20:48,448
LGA,OMA,11:07,13:24,171   LGA,ORD,18:33,20:22,143  LGA,BOS,10:33,12:03,74    LGA,DAL,18:44,22:42,351
OMA,LGA,11:08,13:07,175   ORD,LGA,18:48,21:45,246  BOS,LGA,11:16,13:29,83    DAL,LGA,18:26,21:29,464
LGA,OMA,12:31,14:02,234   LGA,ORD,19:32,21:25,160  LGA,BOS,12:08,14:05,142   LGA,DAL,19:57,23:15,512
OMA,LGA,12:18,14:56,172   ORD,LGA,19:50,22:24,269  BOS,LGA,12:34,15:02,109   DAL,LGA,20:07,23:27,473
LGA,OMA,14:05,15:47,226   LGA,MIA,6:33,9:14,172    LGA,BOS,13:39,15:30,74    LGA,CAK,6:58,9:01,238
OMA,LGA,13:37,15:08,250   MIA,LGA,6:25,9:30,335    BOS,LGA,13:40,15:37,138   CAK,LGA,6:08,8:06,224
LGA,OMA,15:07,17:21,129   LGA,MIA,8:23,11:07,143   LGA,BOS,15:25,16:58,62    LGA,CAK,8:19,11:16,122
OMA,LGA,15:03,16:42,135   MIA,LGA,7:34,9:40,324    BOS,LGA,15:27,17:18,151   CAK,LGA,8:27,10:45,139
LGA,OMA,16:35,18:56,144   LGA,MIA,9:25,12:46,295   LGA,BOS,17:03,18:03,103   LGA,CAK,9:58,12:56,249
OMA,LGA,16:51,19:09,147   MIA,LGA,9:15,12:29,225   BOS,LGA,17:11,18:30,108   CAK,LGA,9:15,12:14,247
LGA,OMA,18:25,20:34,205   LGA,MIA,11:08,14:38,262  LGA,BOS,18:24,20:49,124   LGA,CAK,10:32,13:16,139
OMA,LGA,18:12,20:17,242   MIA,LGA,11:28,14:40,248  BOS,LGA,18:34,19:36,136   CAK,LGA,10:53,13:36,189
LGA,OMA,20:05,21:44,172   LGA,MIA,12:37,15:05,170  LGA,BOS,19:58,21:23,142   LGA,CAK,12:01,13:41,267
OMA,LGA,20:05,22:06,261   MIA,LGA,12:05,15:30,330  BOS,LGA,20:17,22:22,102   CAK,LGA,12:08,14:59,149
LGA,ORD,6:03,8:43,219     LGA,MIA,14:08,16:09,232  LGA,DAL,6:09,9:49,414     LGA,CAK,13:37,15:33,142
ORD,LGA,6:05,8:32,174     MIA,LGA,14:01,17:24,338  DAL,LGA,6:12,10:22,230    CAK,LGA,13:40,15:38,137
LGA,ORD,7:50,10:08,164    LGA,MIA,15:23,18:49,150  LGA,DAL,7:57,11:15,347    LGA,CAK,15:50,18:45,243
ORD,LGA,8:25,10:34,157    MIA,LGA,15:34,18:11,326  DAL,LGA,7:53,11:37,433    CAK,LGA,15:23,17:25,232
LGA,ORD,9:11,10:42,172    LGA,MIA,16:50,19:26,304  LGA,DAL,9:49,13:51,229    LGA,CAK,16:33,18:15,253
ORD,LGA,9:42,11:32,169    MIA,LGA,17:07,20:04,291  DAL,LGA,9:08,12:12,364    CAK,LGA,17:08,19:08,262
LGA,ORD,10:33,13:11,132   LGA,MIA,18:07,21:30,355  LGA,DAL,10:51,14:16,256   LGA,CAK,18:17,21:04,259
ORD,LGA,11:01,12:39,260   MIA,LGA,18:23,21:35,134  DAL,LGA,10:30,14:57,290   CAK,LGA,18:35,20:28,204
LGA,ORD,12:08,14:47,231   LGA,MIA,20:27,23:42,169  LGA,DAL,12:20,16:34,500   LGA,CAK,19:46,21:45,214
ORD,LGA,12:44,14:17,134   MIA,LGA,19:53,22:21,173  DAL,LGA,12:19,15:25,342   CAK,LGA,20:30,23:11,114
```

接下来开始思考如何设计程序，可以把这些航班数据放在字典里，既然每一行有 5 个数据并按逗号分隔，把它按行读取下来。建立一个字典，其中，key 是出发点和终点组成的二元组，value 是所有可能的航班详情组成的一个列表，就像这样：

```
{('LGA', 'CAK'): [('6:58', '9:01', 238), ('8:19', '11:16', 122), ('9:58', '12:56', 249), ('10:32', '13:16', 139), ('12:01', '13:41', 267), ('13:37', '15:33', 142), ('15:50', '18:45', 243), ('16:33', '18:15', 253), ('18:17', '21:04', 259), ('19:46', '21:45', 214)], …..}
```

为了准确表达问题的解，通用的方法是用一个数字的列表，因为每个人都要往返，所以列表的长度是人数的 2 倍，比如[1,4,3,2,…..]的意思是，Seymour 坐当天第 2 次航班从波士顿飞到纽约，然后坐第 5 次航班返回；Franny 坐当天第 4 次航班从达拉斯飞到纽约，然后坐第 3 次航班返回

需要解决的问题就是怎样安排航班使得成本最低，因此，一个能精确描述的成本函数就是优化问题的关键，任何优化算法的目标就是要寻找一组能够使成本函数值最小的输入。在这里成本需要综合考虑航班的价格和人的等待时间

下面介绍并实现两种优化算法：

模拟退火算法(Simulated Annealing)是受物理学启发而来的，退火是指合金被加热后又冷却的过程，大量的原子在这个过程中能够找到一个低能阶的配置并冷却下来

从问题的一个随机解开始，用一个变量来表示温度，温度一开始时很高，然后逐渐变低，每一次迭代期间随机选中解中的某个数字，然后朝某一方向变化。在这个问题中，Seymour 的返程航班也许会从第 2 趟移到第 3 趟，其成本的改变会被计算出来。退火的开始阶段会接受较差的解，随着退火过程的不断进行，算法越来越不可能接受较差的解

Domain 是一个由二元组构成的列表，它指定了列表中每个元素的最大最小值，解的长度和这个列表的长度相同，在本问题中，每个人都有 10 个往程和 10 个返程，因此列表中的 domain 是(0,9)，每个人重复 2 次

为了退火，首先要创建一个具有合适长度的随机解，其中的所有值都在指定的范围内，温度和冷却率是可调的参数，每次迭代时，i 被设置为解的一个随机索引，并将 dir 设为介于 -step 和 step 之间的某个随机数，算法会计算当前的成本以及以 dir 为增量对 i 处的值进行修改时的成本

遗传算法(Genetic Algorithms)是受生物学启发而来的，这类算法的运行过程是先随机生成一组解，我们称之为种群。在优化过程中的每一步，算法会计算整个种群的成本函数，从而得到一个有关题解及其成本的有序列表

在对题解进行排序后，一个新的种群——我们称之为下一代——被创建出来了，首先，把当前这一代中位于最顶端的题解加入到新的种群中，这个过程称为精英选拔，新种群中的余下部分是由修改最优解后形成的全新解所组成的

有两种修改题解的方法，变异：通常的实现是对一个既有解做微小、简单、随机的改变，在本问题中，实现方式是在题解中选择一个数字，然后把它递增或递减即可。交叉：选取最优解中的两个解，然后按某种方式将它们结合，在本问题中，实现交叉的简单做法是，从一个解中随机取出一些数字作为新的题解(孩子)中的某个元素，而剩余元素来自另一个题解

一个新的种群是通过对最优解进行随机的变异和交叉处理构造出来的，它的大小通常和旧的种群相同

通俗地解释一下遗传算法：

1. 首先我们设定好了国民的初始人群数量
2. 然后我们定义了一个函数用来区分好人和坏人
3. 再次我们选择出好人并且让他们繁衍自己的后代
4. 最后这些后代们从原来的国民中替代了部分坏人并且不断重复这一过程

```
#coding=utf-8
import time
import random
import math
people = [('Seymour', 'BOS'), ('Franny', 'DAL'), ('Zooey', 'CAK'), ('Walt', 'MIA'),
('Buddy', 'ORD'), ('Les', 'OMA')]
destination = 'LGA'
flights = {}
for line in file('schedule.txt'):
    origin, dest, depart, arrive, price = line.strip().split(',')
    flights.setdefault((origin, dest), [])
                                                    # 将航班详情添加到航班列表中
    flights[(origin, dest)].append((depart, arrive, int(price)))
                                                    #计算某个给定时间在一天中的分钟数
def getminutes(t):
    x = time.strptime(t, '%H:%M')
    return x[3] * 60 + x[4]
                                                    #如何准确描述问题的解
def printschedule(r):
    for d in range(len(r) / 2):
        name = people[d][0]
        origin = people[d][1]
        out = flights[(origin, destination)][int(r[d])]
        ret = flights[(destination, origin)][int(r[d + 1])]
        print '%10s%10s %5s-%5s $%3s %5s-%5s $%3s' % (name, origin, out[0], out[1], out[2],
ret[0], ret[1], ret[2])
                                                    #人名和起点、出发和到达时间、往返航班的票价
                                                    #成本函数
def schedulecost(sol):
    totalprice = 0
```

```
    latestarrival = 0
    earliestdep = 24 * 60
    for d in range(len(sol) / 2):
                                        # Get the inbound and outbound flights
                                           #得到往程航班和返程航班
        origin = people[d][1]
        outbound = flights[(origin, destination)][int(sol[d])]
        returnf = flights[(destination, origin)][int(sol[d + 1])]
                        # Total price is the price of all outbound and return flights
        totalprice += outbound[2]
        totalprice += returnf[2]
                                              #记录最晚到达时间和最早离开时间
                                       # Track the latest arrival and earliest departure
        if latestarrival < getminutes(outbound[1]): latestarrival =
getminutes(outbound[1])
        if earliestdep > getminutes(returnf[0]): earliestdep = getminutes(returnf[0])
                    # Every person must wait at the airport until the latest person arrives.
                        # They also must arrive at the same time and wait for their flights.
    totalwait = 0
    for d in range(len(sol) / 2):
        origin = people[d][1]
        outbound = flights[(origin, destination)][int(sol[d])]
        returnf = flights[(destination, origin)][int(sol[d + 1])]
        totalwait += latestarrival - getminutes(outbound[1])
        totalwait += getminutes(returnf[0]) - earliestdep
                    # Does this solution require an extra day of car rental? That'll be $50!
    if latestarrival > earliestdep: totalprice += 50
    return totalprice + totalwait
                                                                    #模拟退火算法
def annealingoptimize(domain, costf, T=10000.0, cool=0.95, step=1):
                                         # Initialize the values randomly,随机初始化值
    vec = [float(random.randint(domain[i][0], domain[i][1]))
           for i in range(len(domain))]
    while T > 0.1:
                                           # Choose one of the indices 选择一个索引值
        i = random.randint(0, len(domain) - 1)
                                # Choose a direction to change it 选择一个改变索引值的方向
        dir = random.randint(-step, step)
                                        #创建一个代表问题的解的新列表，改变其中一个值
                                       # Create a new list with one of the values changed
        vecb = vec[:]
        vecb[i] += dir
        if vecb[i] < domain[i][0]:
            vecb[i] = domain[i][0]
        elif vecb[i] > domain[i][1]:
            vecb[i] = domain[i][1]
                        # Calculate the current cost and the new cost 计算当前成本和新的成本
        ea = costf(vec)
        eb = costf(vecb)
        p = pow(math.e, (-eb - ea) / T)
                                     # Is it better, or does it make the probability cutoff?
                                      # 它是更好的解吗？或者是趋向最优解的可能的临界解吗？
        if (eb < ea or random.random() < p):
            vec = vecb
                                                          # Decrease the temperature
        T = T * cool
    return vec
                                                                #遗传算法，
                        #popsize:种群大小      mutprob:种群的新成员是由变异而非交叉得来的概率
                    #maxiter:需运行多少代  elite:种群中被认为是优解且被允许传入下一代的部分
def geneticoptimize(domain, costf, popsize=50, step=1,mutprob=0.2, elite=0.2,
maxiter=100):
                                            # Mutation Operation 变异操作
```

```
    def mutate(vec):
        i = random.randint(0, len(domain) - 1)
        if random.random() < 0.5 and vec[i] > domain[i][0]:
            return vec[0:i] + [vec[i] - step] + vec[i + 1:]
        elif vec[i] < domain[i][1]:
            return vec[0:i] + [vec[i] + step] + vec[i + 1:]
                                                    # Crossover Operation 交叉操作
    def crossover(r1, r2):
        i = random.randint(1, len(domain) - 2)
        return r1[0:i] + r2[i:]
                                                # Build the initial population 构造初始种群
    pop = []
    for i in range(popsize):
        vec = [random.randint(domain[i][0], domain[i][1])
              for i in range(len(domain))]
        pop.append(vec)
                                    # How many winners from each generation?每一代中有多少胜出者?
    topelite = int(elite * popsize)
                                                                        # Main loop 主循环
    for i in range(maxiter):
        scores = [(costf(v), v) for v in pop]
        scores.sort()
        ranked = [v for (s, v) in scores]
                                            # Start with the pure winners 从纯粹的胜出者开始
        pop = ranked[0:topelite]
                                                # Add mutated and bred forms of the winners
                                                                #添加变异和交叉后的胜出者
        while len(pop) < popsize:
            if random.random() < mutprob:
                                                                        # Mutation
                c = random.randint(0, topelite)
                pop.append(mutate(ranked[c]))
            else:
                                                                        # Crossover
                c1 = random.randint(0, topelite)
                c2 = random.randint(0, topelite)
                pop.append(crossover(ranked[c1], ranked[c2]))
                                                # Print current best score 打印当前最优解
        print scores[0][0]
    return scores[0][1]
```

把以上代码保存到脚本 optimization.py 中，在这个脚本所在的文件夹中新建一个.py 文件调用它

```
import optimization
domain = [(0, 9)]*(len(optimization.people)*2)
s1 = optimization.annealingoptimize(domain, optimization.schedulecost)
print optimization.schedulecost(s1)
optimization.printschedule(s1)
s2 = optimization.geneticoptimize(domain, optimization.schedulecost)
print optimization.schedulecost(s2)
print s2
optimization.printschedule(s2)
```

退火过程的结果：每一次运行都会不相同

```
2932
   Seymour      BOS 11:16-13:29 $ 83 10:33-12:03 $ 74
    Franny      DAL 10:30-14:57 $290 10:51-14:16 $256
     Zooey      CAK 10:53-13:36 $189 10:32-13:16 $139
      Walt      MIA 11:28-14:40 $248 12:37-15:05 $170
     Buddy      ORD 12:44-14:17 $134 14:19-17:09 $190
       Les      OMA 13:37-15:08 $250 11:07-13:24 $171
```

遗传算法的运行经常报错：

```
    scores = [(costf(v), v) for v in pop]
    returnf = flights[(destination, origin)][int(sol[d + 1])]
IndexError: list index out of range
```

连续尝试点击运行几十次之后才成功一次：

```
3476    3273    3103    3065    3065    2992    2992    2868    2868..  2868..  2868
3476    3273    3103    3065    2992    2992    2992    2868..  2868..  2868
3273    3235    3103    3065    2992    2992    2992    2868..  2868..  2868
[7, 6, 7, 6, 6, 7, 5, 0, 0, 0, 0, 0]
   Seymour      BOS 17:11-18:30 $108 15:25-16:58 $ 62
    Franny      DAL 15:44-18:55 $382 17:14-20:59 $277
     Zooey      CAK 17:08-19:08 $262 15:50-18:45 $243
      Walt      MIA 15:34-18:11 $326 15:23-18:49 $150
     Buddy      ORD 15:58-18:40 $173 17:06-20:00 $ 95
       Les      OMA 16:51-19:09 $147 15:07-17:21 $129
```

机器学习(Machine Learning)是 AI 领域中与算法相关的一个子领域，计算机学习的过程相当于将一组数据传递给我们在计算机上编写的算法，而这些人为的非随机的数据往往是包含了“模式(patterns)”的，不断的重复训练之后计算机能够归纳出模式的相关信息，于是它就能利用这些信息来预测未来可能出现的数据。可见，许多机器学习算法都依赖于数学和概率统计的知识

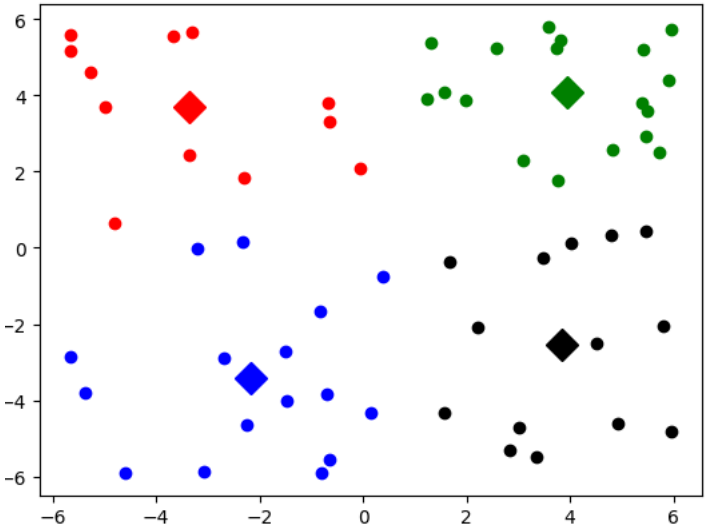
利用带有标准正确答案的样本输入数据和期望输出训练如何预测的方法叫做监督学习法；不需要人类经验干预的、在一组无标签的数据中自动寻找某种相似结构并且分类的过程叫做无监督学习

聚类(clustering)是无监督学习的一个例子，其中一种聚类算法是 **K-均值聚类**(K-Means)在这个算法中，我们会预先告诉算法希望生成的聚类数量，然后算法会根据数据的结构状况来确定聚类的大小。首先随机确定 K 个质心，然后对所有的点求到这些质心的距离，假如某点离某质心最近，那么该点和该质心组成该质心的点群，把该质心移动到它的点群中心。不断重复第 2 步和第 3 步，直到质心不再移动为止。只要把现实事物的属性用向量来抽象表达，就可以使用这个算法来实现“物以类聚”

25.二维可视化聚类.py

```
# coding=utf-8
from numpy import *
dataset = []
for i in range(60):
    dataset.append([random.uniform(-6, 6), random.uniform(-6, 6)])
dataset = array(dataset)
# 计算两个向量的距离，用的是欧几里得距离
def distEclud(vecA, vecB):
    return sqrt(sum(power(vecA - vecB, 2)))
# 随机生成初始的质心（ng 的课说的初始方式是随机选 K 个点）
def randCent(dataset, k):
    n = shape(dataset)[1]
    centroids = mat(zeros((k, n)))
    for j in range(n):
        minJ = min(dataset[:, j])
        rangeJ = float(max(dataset[:, j]) - minJ)
        centroids[:, j] = minJ + rangeJ * random.rand(k, 1)
    return centroids
def kMeans(dataset, k, distMeas=distEclud, createCent=randCent):
    m = shape(dataset)[0]
    clusterAssment = mat(zeros((m, 2)))          # create mat to assign data points
                                                 # to a centroid, also holds SE of each point
    centroids = createCent(dataset, k)
    clusterChanged = True
    while clusterChanged:
        clusterChanged = False
        for i in range(m):
                                # for each data point assign it to the closest centroid
            minDist = inf
            minIndex = -1
            for j in range(k):
                distJI = distMeas(centroids[j, :], dataset[i, :])
```

```
                if distJI < minDist:
                    minDist = distJI
                    minIndex = j
            if clusterAssment[i, 0] != minIndex:
                clusterChanged = True
            clusterAssment[i, :] = minIndex, minDist ** 2
        print centroids
        for cent in range(k):                # recalculate centroids
            ptsInClust = dataset[nonzero(clusterAssment[:, 0].A == cent)[0]]
                                             # get all the point in this cluster
            centroids[cent, :] = mean(ptsInClust, axis=0)
                                                 # assign centroid to mean
    return centroids, clusterAssment
def show(dataset, k, centroids, clusterAssment):
    from matplotlib import pyplot as plt
    numSamples, dim = dataset.shape
    mark = ['or', 'ob', 'og', 'ok', '^r', '+r', 'sr', 'dr', '<r', 'pr']
    for i in xrange(numSamples):
        markIndex = int(clusterAssment[i, 0])
        plt.plot(dataset[i, 0], dataset[i, 1], mark[markIndex])
    mark = ['Dr', 'Db', 'Dg', 'Dk', '^b', '+b', 'sb', 'db', '<b', 'pb']
    for i in range(k):
        plt.plot(centroids[i, 0], centroids[i, 1], mark[i], markersize=12)
    plt.show()
def main():
    myCentroids, clustAssing = kMeans(dataset, 4)
    print myCentroids
    show(dataset, 4, myCentroids, clustAssing)
if __name__ == '__main__':
    main()
```



Python 中有一个常用的机器学习模块 sklearn(scikit-learn)
但是这里我打算自己实现一些机器学习算法的最基本版，而不是调用这个模块

神经网络(Neural Networks)是一种监督学习法，它非常类似于应试教育。我们可以在计算机中搭建一个人工神经网络，来模拟人脑中的思考过程。为了简单起见，这里只模拟一个神经元，它有 3 个输入和 1 个输出

训练这个神经元来解决下面这个问题(右图)，前面 4 个数据称为“训练集”，你能求解出模式吗？“？”处应该是 0 还是 1 呢？

	Input			Output
Example 1	0	0	1	0
Example 2	1	1	1	1
Example 3	1	0	1	1
Example 4	0	1	1	0

New situation	1	0	0	**?**

这里设定的标准答案是：输出总是等于第一列的输入，右边的？应该是 1

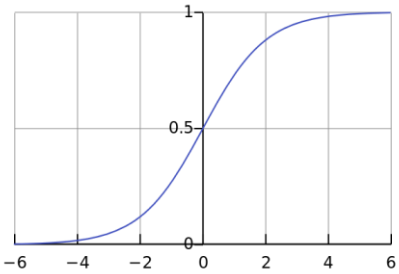
现在要做的，是在计算机上模拟神经网络，通过这些训练集(就像做题一样，这是平时的习题集，有标准答案的)让计算机“学习”并发现这个规律(模式)，然后正确解决问题(参加高考)

那么应该怎么做？人脑解决这个问题时，会观察 3 个输入，最后总结出规律：后面两个输入是不需要的。也就是说，每个输入都有一个权重，在计算机中模拟时可以设定这个权重，权重的绝对值代表输入对输出的决定权，在开始之前，先把权重设为随机数，再开始训练过程

1. 从训练集样本读取输入，根据权重进行调整，再设计并代入某个特殊的方程计算神经元的输出
2. 计算误差，也就是神经元的实际输出和训练样本的期望输出之差
3. 根据误差的方向，微调权重
4. 重复 10000 次

那么，这个特殊方程应该怎么设计？因为输出在 0 到 1 之间，所以这里可以用一个很方便的激活函数：$\frac{1}{1+e^{-x}}$

$$\text{output} = \frac{1}{1+e^{-(\sum weight_i \times input_i)}}$$



我们还需要一个合适的调整权重的方程，首先我们希望调整量和误差量成正比，然后再乘以输入，如果输入为 0，权重就不会被调整，最后乘以 Sigmoid 曲线的梯度(这个函数的优点在于，它在绝对值较大的两端处变化率较小，如果神经元确定当前权重值是正确的，就不需要太大调整，乘以这个曲线的梯度即可)

误差加权导数方程：调整量 = 误差 × 输入 × 用来计算输出的函数的梯度

曲线梯度可以用隐函数求导得到：$y = \frac{1}{1+e^{-x}}$　　$y' = y(1-y)$

最终的权重调整方程就是：调整量 = 误差 × 输入 × 输出 × (1 − 输出)

26.简单的人工神经网络.py

```
from numpy import exp, array, random, dot
training_set_inputs = array([[0, 0, 1], [1, 1, 1], [1, 0, 1], [0, 1, 1]])
training_set_outputs = array([[0, 1, 1, 0]]).T
random.seed(1)
synaptic_weights = 2*random.random((3, 1))-1
for i in xrange(10000):
    output = 1/(1+exp(-(dot(training_set_inputs, synaptic_weights))))
    synaptic_weights += dot(training_set_inputs.T, (training_set_outputs -
output)*output*(1-output))
print 1/(1+exp(-dot(array([1, 0, 0]), synaptic_weights)))
```

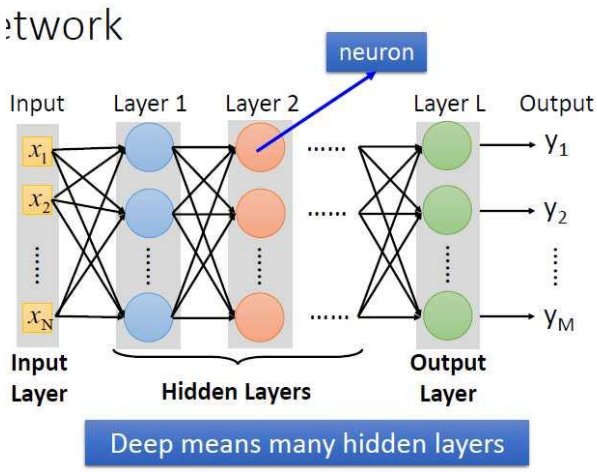
输出结果：

```
[ 0.99993704]
```

如果去掉代码中设置随机数种子的那一句，每次输出结果会略有不同

神经网络几乎总是利用 S 型函数来计算神经元的输出，另一种常用的 S 型函数是 tanh（反双曲正切）

神经网络可以有多个中间层，因为外界无法直接与其交互，所以也被称为隐藏层，它负责对输入进行组合



深度学习(Deep Learing)是采用神经网络，用于解决线性不可分的问题，所谓深度，意思是具有多个隐层

接下来，我们根据一个人为假设的简单模型来构造一个有关葡萄酒价格的数据集，酒的价格由等级和储藏年代共同决定，该模型假设葡萄酒存在“峰值年”现象，并且对原有价格随机加减以表现税收或价格变动等不稳定因素。

对于这个葡萄酒定价问题，简单的做法是找到几瓶情况最为相近的酒，并假设其价格大体相同，算法通过寻找与当前所关注的商品情况相似的一组商品，对它们的价格求均值，进而做出价格预测，这种方法称为**K-最近邻算法**(K-nearest neighbors,kNN)

这里的 k 是指为了求得最终结果而参与求平均的商品数目，为了减少“噪声”的干扰，最好多选取一些近邻，扩大信息收集面。首先要衡量两个商品之间的相似程度，我们简单地采用了欧几里得距离，该函数在计算距离时对葡萄酒的年代和等级同等看待，现实情况很少这样做。我们还需要一个函数来计算给定商品与原数据集中其他任意商品的距离，把这些距离存进一个列表并排序，算法将对这个列表的前 k 项求平均。kNN 的计算量很大，但是每次有新数据加入时都无需重新训练。

事实上简单求平均是不够好的，要根据距离的远近来分配权重，在这里我选择了高斯权重函数，距离为 0 时权重为 1，距离增大时权重趋近于 0

27.葡萄酒的价格.py

```
from random import random
import math
def wineprice(rating, age):
    peak_age = rating - 50
    price = rating/2
    if age > peak_age:
        price = price*(5-(age-peak_age))
    else:
        price = price*(5*((age+1)/peak_age))
    if price < 0:
        price = 0
    return price
def wineset1():
    rows = []
    for i in range(300):
        rating = random()*50+50
        age = random()*50
        price = wineprice(rating, age)
        price *= (random()*0.4+0.8)
        rows.append({'input': (rating, age), 'result': price})
    return rows
def euclidean(v1, v2):
    d = 0.0
    for i in range(len(v1)):
        d += (v1[i]-v2[i])**2
    return math.sqrt(d)
def getdistance(data, vec1):
    distancelist = []
    for i in range(len(data)):
        vec2 = data[i]['input']
        distancelist.append((euclidean(vec1, vec2), i))
    distancelist.sort()
    return distancelist
def gaussian(dist, sigma = 10.0):
    return math.e**(-dist**2/(2*sigma**2))
def weightedknn(data, vec1, k=5, weightf = gaussian):
    dlist = getdistance(data, vec1)
    avg = 0.0
    totalweight = 0.0
    for i in range(k):
        dist = dlist[i][0]
        idx = dlist[i][1]
        weight = weightf(dist)
        avg += weight*data[idx]['result']
```

```
        totalweight += weight
    avg = avg/totalweight
    return avg
data = wineset1()
print weightedknn(data, (99.0, 5.0))
```

输出结果：31.2211912739

机器学习常用于分类问题，其中一个例子是过滤垃圾邮件和垃圾信息。在这里我们简单模拟一下这个过程，对一些简单的句子做分类，如果这个句子是垃圾短信，就过滤掉

分类器需要利用某些特征来进行分类，当对文档进行分类时，内容就是文档，而特征是文档中的单词。把单词作为特征时，其假设是：某些单词相对而言更有可能出现在垃圾信息中。这一假设是大多数垃圾信息过滤器依赖的基本前提

所以就需要一个函数，它负责取出文档中的单词并保存起来，这里选用字典来保存，为了简化问题，可以只保存不重复的单词。如何正确选择、定义、提取特征是首先要完成的，这个函数命名为 getwords()

可以把分类器做成一个类，如果有不同的分类方法，就再建一个类来继承它。这个基本的父类命名为 classifier，现在可以定义变量了，我们需要表示特征(f，即单词)、类型(cat，分为 good 和 bad)。在这个类中设计 3 个实例变量：fc： 记录位于各个类型中的不同特征的数量，我们可以用这样一个字典来存储它：fc = {'python': {'bad': 0,'good': 6}, 'the': {'bad': 3, 'good': 3}}；cc：记录各个类型被使用的次数，同样可以用字典；getfeatures：负责提取特征的函数，对应于 getwords

分类器的训练就是把一组训练用的样本数据传给分类器(这个函数命名为 sampletrain)，然后让分类器对 fc、cc 不断更新，这个更新过程包括增加和统计计数值，这些过程都应该写成方法封装在父类中

接下来就是把频率转化成概率，比较某个单词出现在各种类型中的概率，我们才能正确分类对于一个给定分类（条件），某个单词出现的概率就是条件概率 P(word|classification)。计算这个概率只需把单词在某个分类中出现的次数除以该分类中的单词总数，该函数命名为 fprob。这样计算出的结果是比较偏激的，一个合理的改进方法是，假设概率的初始值为 0.5，并且给这个假设值赋予一个权重，权重为 1 表示它相当于一个单词，用这样的方法计算加权后的概率：

$$(\text{权重} \times \text{假设的初始概率 } 0.5 + \text{训练期间统计单词在所有类型中的出现次数 count} \times \text{其中一次单词在某个类型中出现的概率 } 1.0 \text{ 或者 } 0) \div (\text{count} + \text{权重})$$

比如单词 money，它一开始的加权概率对所有分类而言都是 0.5，然后训练期间不断接收 money 出现在某个分类中，这个加权概率就能不断调整。

既然求出了指定单词在某个分类中出现的概率，现在就应该思考如何把一个句子中的所有单词组合起来得到这个句子属于某个分类的概率，这样才能对句子进行分类。接下来我们实现**朴素贝叶斯(Naïve Bayes)**分类器，这个算法的 naïve 之处在于，它假设将要被组合的各个概率之间是相互独立的，即一个单词出现在某个分类中的概率和其他单词出现在这个分类中的概率是不相关的。显然这个假设事实上不成立，比如包含单词 casino 的文档更有可能包含单词 money。不过我们还是可以对各个分类的计算结果进行比较然后看哪个分类的概率最大，尽管这个概率已经不是真实的概率。在现实应用中朴素贝叶斯被证明是非常有效的

建立一个子类 naïvebayes，在里面定义一个方法 docprob 将所有特征的概率相乘(正是利用了“朴素”的假设)得到整体的概率，即 P(document|category)

但是，我们真正需要的概率是：P(category|document)，即对于一个给定的文档，它属于某个分类的概率是多少？这里就需要借助贝叶斯定理：

P(category|document) = P(document|category)×P(category)/P(document)

P(document|category)前面已经求出来了，P(category)等于属于某一分类的文档数除以文档的总数。因为不管计算的是哪个分类，P(document)的值都是一样的，所以这一项可以直接忽略

最后一步考虑到的问题是，我们可以容忍收到垃圾短信，但是绝对不能容忍收不到正常短信，因此我们需要一个阈值，比如阈值为 3，意思是针对 bad 分类的概率必须超过针对 good 分类概率的 3 倍时，才能把这个短信归类为垃圾。定义一个 classify 方法，该方法计算每一个分类的概率，得到最大值，确保它大于阈值乘以第二大概率值，如果不能确保，就返回默认值。在这里的简单分类中，只有两类 good 和 bad，如果 good 是最大的，bad 就是第二大

28.句子分类.py

```
#coding=utf-8
import re
def sampletrain(cl):            #将训练集导入分类器
    cl.train('Nobody owns the water.', 'good')
    cl.train('the quick rabbit jumps fences', 'good')
    cl.train('buy pharmaceuticals now', 'bad')
    cl.train('make quick money at the online casino', 'bad')
    cl.train('the quick brown fox jumps', 'good')
def getwords(doc):
    splitter = re.compile('\\W*')      #根据非字母字符进行单词拆分
    words = [s.lower() for s in splitter.split(doc) if len(s)>2 and len(s)<20]
    return dict([(w, 1) for w in words])
                                       #只返回一组不重复的转成小写形式的单词
class classifier:
    def __init__(self,getfeatures,filename=None):
        self.fc = {}
        self.cc = {}
        self.getfeatures = getfeatures
    def incf(self,f,cat):              #增加对特征/分类组合的计数值
        self.fc.setdefault(f, {})
        self.fc[f].setdefault(cat, 0)
        self.fc[f][cat] += 1
    def incc(self,cat):                #增加对某一分类的计数值
        self.cc.setdefault(cat, 0)
        self.cc[cat] += 1
    def fcount(self,f,cat):            #某一特征出现于某一分类中的次数
        if f in self.fc and cat in self.fc[f]:
            return float(self.fc[f][cat])
        return 0.0
    def catcount(self,cat):            #属于某一分类的内容项数量
        if cat in self.cc:
            return float(self.cc[cat])
        return 0
    def totalcount(self):              #所有内容项的数量
        return sum(self.cc.values())
    def categories(self):              #所有分类的列表
        return self.cc.keys()
    def train(self,item,cat):
        features = self.getfeatures(item)
        for f in features:
            self.incf(f, cat)       #针对该分类为每个特征增加计数值
        self.incc(cat)             #增加针对该分类的计数值
    def fprob(self,f,cat):
        if self.catcount(cat)==0:
            return 0
        return self.fcount(f, cat)/self.catcount(cat)
    def weightprob(self,f,cat,prf,weight=1.0,ap=0.5):
        basicprob = prf(f, cat)
        totals = sum([self.fcount(f, c) for c in self.categories()])
        bp = ((weight*ap)+(totals*basicprob))/(weight+totals)
        return bp                      #计算并返回加权平均
```

```
class naivebayes(classifier):
    def __init__(self,getfeatures):
        classifier.__init__(self,getfeatures)
        self.thresholds = {}                                    #定义阈值
    def docprob(self,item,cat):
        features = self.getfeatures(item)
        p = 1
        for f in features:
            p *= self.weightprob(f, cat, self.fprob)
        return p
    def prob(self,item,cat):
        catprob = self.catcount(cat)/self.totalcount()
        docprob = self.docprob(item, cat)
        return docprob*catprob
    def setthreshold(self,cat,t):
        self.thresholds[cat] = t
    def getthreshold(self,cat):                                 #取阈值
        if cat not in self.thresholds:
            return 1.0
        return self.thresholds[cat]
    def classify(self, item, default=None):
        probs = {}                  # Find the category with the highest probability
        maxl = 0.0
        for cat in self.categories():
            probs[cat] = self.prob(item, cat)
            if probs[cat] > maxl:
                maxl = probs[cat]
                best = cat
                                # Make sure the probability exceeds threshold*next best
        for cat in probs:
            if cat == best: continue
            if probs[cat] * self.getthreshold(best) > probs[best]: return default
        return best
cl = classifier(getwords)
sampletrain(cl)
print cl.weightprob('money', 'good', cl.fprob)
sampletrain(cl)
print cl.weightprob('money', 'good', cl.fprob)
cl = naivebayes(getwords)
cl.setthreshold('bad', 3.0)
for i in range(10):
    sampletrain(cl)
    print cl.classify('quick money', default='unknown'),
print getwords('python is a dynamic language pytho ghh')
```

运行结果：

```
0.25
0.166666666667
unknown unknown unknown unknown bad bad bad bad bad bad {'python': 1, 'ghh': 1, 'dynamic':
1, 'pytho': 1, 'language': 1}
```

当一个很受欢迎的网站提供免费账号和付费账号时，预测一位用户成为付费用户的可能性就成为一项有价值的工作。这也是机器学习的一种应用，当计算机学习到哪些因素使免费用户变成付费用户时，它就能以此为依据来指导网站的广告策略。

可以在服务器的日志中收集用户的痕迹信息比如：来源网站、地理位置、是否阅读过 FAQ、浏览网页数。然后我们看这些用户中有哪些后来变成了付费用户，把这些样本输入给计算机，让它训练自己，然后给计算机输入某一个用户的痕迹信息，让计算机预测该用户是否可能成为付费用户。

把这些信息存放到一个二维的列表中，列表中每一个元素(每一行)代表一个用户，每一列代表一种属性(比如浏览网页数)，这个列表命名为 my_data

接下来实现**决策树算法**(Decision Tree)，算法思想很简单，它直接按照属性拆分样本数据，你只需要通过观察就能理解其推导的过程

建立一个名为 decisionnode 的类，表示树上的每一个结点，每一个结点包含 5 个实例变量：col:待检验的判断条件所对应的列索引值，value:为了使结果为真，当前列必须匹配的值，tb,fb:分别对应结果为真或假时的子树上的孩子结点，results:保存针对当前分支的结果，是一个字典，除叶子结点外它的值都是 None

首先我们要在单独的一列上拆分样本数据集，我们把这个函数命名为 divideset，它根据数据集中某一列属性把数据集拆成由两个数据集构成的元组。数据集中有数值型也有字符串型，需要分类讨论。如果是数值型，就比较它和 value 哪个大；如果是字符串型，就看它是否和 value 相等

一次拆分的结果往往是不理想的，因为用户是否成为付费用户并不是由单一属性决定的，并非浏览网页数超过 21 的用户就一定都变成付费用户而跟地理位置没关系，初次拆分结果是混乱的

但是不同属性拆出来的混乱程度不一样，我们可以计算和比较这些属性的拆分结果的混乱度，然后从能够使混乱度最小的那个属性开始拆起，因为这样能避免很多重复计算，减少决策树的深度。于是需要一个函数 uniquecounts 来对每一项结果计数。既然熵是系统混乱度的度量，那就写一个函数 entropy 来计算熵

然后建立决策树，建树自然是用递归的方式，为了弄明白一个属性的好坏程度，首先求出整个数据集的熵，然后尝试利用每一个属性来拆数据集，同时求出每两个新数据集的熵。为了从能够使混乱度最小的那个属性开始拆起，就必须选择信息增益(拆分之前的熵减去拆分之后两个熵的加权平均)最大的那个属性。这样就获得了最佳拆分条件，然后按这个条件建立左右子树。再写一个函数 drawtree 把建好的树画出来，它需要计算树的高度和深度并在结点处打印决策的内容。最后是用来预测的函数 classify，它接收一组新的用户数据列表作为参数，利用决策树采用递归方式对其分类。

29.预测注册用户.py

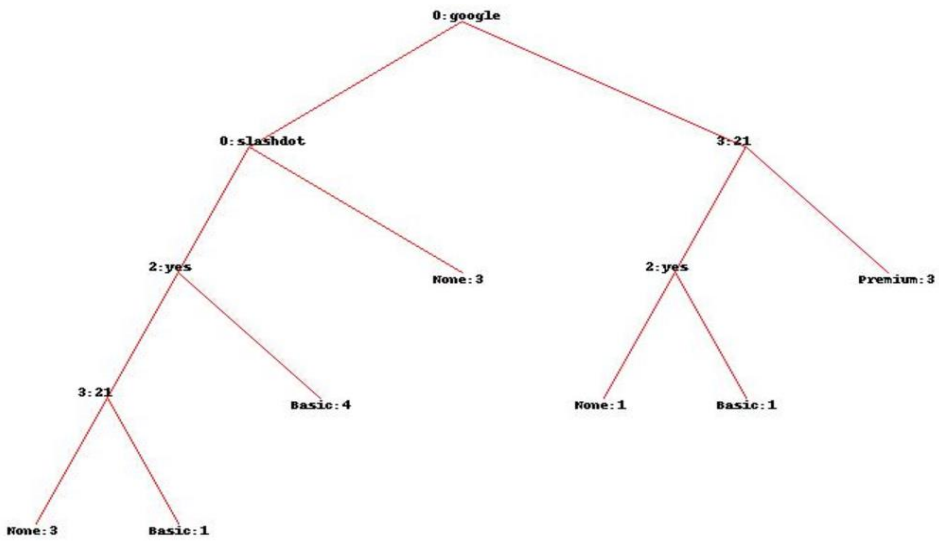
```
my_data = [['slashdot', 'USA', 'yes', 18, 'None'],
          ['google', 'France', 'yes', 23, 'Premium'],
          ['digg', 'USA', 'yes', 24, 'Basic'],
          ['kiwitobes', 'France', 'yes', 23, 'Basic'],
          ['google', 'UK', 'no', 21, 'Premium'],
          ['(direct)', 'New Zealand', 'no', 12, 'None'],
          ['(direct)', 'UK', 'no', 21, 'Basic'],
          ['google', 'USA', 'no', 24, 'Premium'],
          ['slashdot', 'France', 'yes', 19, 'None'],
          ['digg', 'USA', 'no', 18, 'None'],
          ['google', 'UK', 'no', 18, 'None'],
          ['kiwitobes', 'UK', 'no', 19, 'None'],
          ['digg', 'New Zealand', 'yes', 12, 'Basic'],
          ['slashdot', 'UK', 'no', 21, 'None'],
          ['google', 'UK', 'yes', 18, 'Basic'],
          ['kiwitobes', 'France', 'yes', 19, 'Basic']]
class decisionnode:
    def __init__(self, col=-1, value=None, results=None, tb=None, fb=None):
        self.col = col
        self.value = value
        self.results = results
        self.tb = tb
        self.fb = fb
                    # Divides a set on a specific column. Can handle numeric or nominal values
def divideset(rows, column, value):
                                        # Make a function that tells us if a row is in
                                    # the first group (true) or the second group (false)
    split_function = None
    if isinstance(value, int) or isinstance(value, float):
        split_function = lambda row: row[column] >= value
    else:
```

```
        split_function = lambda row: row[column] == value
                                                # Divide the rows into two sets and return them
    set1 = [row for row in rows if split_function(row)]
    set2 = [row for row in rows if not split_function(row)]
    return (set1, set2)
              # Create counts of possible results (the last column of each row is the result)
def uniquecounts(rows):
    results = {}
    for row in rows:
                                                            # The result is the last column
        r = row[len(row) - 1]
        if r not in results: results[r] = 0
        results[r] += 1
    return results
              # Entropy is the sum of p(x)log(p(x)) across all the different possible results
def entropy(rows):
    from math import log
    log2 = lambda x: log(x) / log(2)
    results = uniquecounts(rows)
                                                             # Now calculate the entropy
    ent = 0.0
    for r in results.keys():
        p = float(results[r]) / len(rows)
        ent = ent - p * log2(p)
    return ent
def buildtree(rows, scoref=entropy):
    if len(rows) == 0: return decisionnode()
    current_score = scoref(rows)
                                        # Set up some variables to track the best criteria
    best_gain = 0.0
    best_criteria = None
    best_sets = None
    column_count = len(rows[0]) - 1
    for col in range(0, column_count):
                                  # Generate the list of different values in this column
        column_values = {}
        for row in rows:
            column_values[row[col]] = 1
                                 # Now try dividing the rows up for each value in this column
        for value in column_values.keys():
            (set1, set2) = divideset(rows, col, value)
                                                                 # Information gain
            p = float(len(set1)) / len(rows)
            gain = current_score - p * scoref(set1) - (1 - p) * scoref(set2)
            if gain > best_gain and len(set1) > 0 and len(set2) > 0:
                best_gain = gain
                best_criteria = (col, value)
                best_sets = (set1, set2)
                                                                # Create the sub branches
    if best_gain > 0:
        trueBranch = buildtree(best_sets[0])
        falseBranch = buildtree(best_sets[1])
        return decisionnode(col=best_criteria[0], value=best_criteria[1],
                            tb=trueBranch, fb=falseBranch)
    else:
        return decisionnode(results=uniquecounts(rows))
def getwidth(tree):
    if tree.tb == None and tree.fb == None: return 1
    return getwidth(tree.tb) + getwidth(tree.fb)
def getdepth(tree):
    if tree.tb == None and tree.fb == None: return 0
    return max(getdepth(tree.tb), getdepth(tree.fb)) + 1
from PIL import Image, ImageDraw
def drawtree(tree, jpeg='tree.jpg'):
    w = getwidth(tree) * 100
```

```
    h = getdepth(tree) * 100 + 120
    img = Image.new('RGB', (w, h), (255, 255, 255))
    draw = ImageDraw.Draw(img)
    drawnode(draw, tree, w / 2, 20)
    img.save(jpeg, 'JPEG')
def drawnode(draw, tree, x, y):
    if tree.results == None:
                                                    # Get the width of each branch
        w1 = getwidth(tree.fb) * 100
        w2 = getwidth(tree.tb) * 100
                                                # Determine the total space required by this node
        left = x - (w1 + w2) / 2
        right = x + (w1 + w2) / 2
                                                            # Draw the condition string
        draw.text((x - 20, y - 10), str(tree.col) + ':' + str(tree.value), (0, 0, 0))
                                                           # Draw links to the branches
        draw.line((x, y, left + w1 / 2, y + 100), fill=(255, 0, 0))
        draw.line((x, y, right - w2 / 2, y + 100), fill=(255, 0, 0))
                                                            # Draw the branch nodes
        drawnode(draw, tree.fb, left + w1 / 2, y + 100)
        drawnode(draw, tree.tb, right - w2 / 2, y + 100)
    else:
        txt = ' \n'.join(['%s:%d' % v for v in tree.results.items()])
        draw.text((x - 20, y), txt, (0, 0, 0))
def classify(observation, tree):
    if tree.results != None:
        return tree.results
    else:
        v = observation[tree.col]
        branch = None
        if isinstance(v, int) or isinstance(v, float):
            if v >= tree.value:
                branch = tree.tb
            else:
                branch = tree.fb
        else:
            if v == tree.value:
                branch = tree.tb
            else:
                branch = tree.fb
        return classify(observation, branch)
tree1 = buildtree(my_data)
drawtree(tree1, jpeg='hahaaaa.jpg')
print classify(['(direct)', 'USA', 'yes', 5], tree1)
```

输出结果：

```
{'Basic': 4}
```



UCI 数据库是加州大学欧文分校(University of CaliforniaIrvine)提出的用于机器学习的数据库，这个数据库目前共有 335 个数据集，其数目还在不断增加，UCI 数据集是一个常用的标准测试数据集

```
5.7,2.9,4.2,1.3,Iris-versicolor
6.2,2.9,4.3,1.3,Iris-versicolor
5.1,2.5,3.0,1.1,Iris-versicolor
5.7,2.8,4.1,1.3,Iris-versicolor
6.3,3.3,6.0,2.5,Iris-virginica
5.8,2.7,5.1,1.9,Iris-virginica
7.1,3.0,5.9,2.1,Iris-virginica
6.3,2.9,5.6,1.8,Iris-virginica
6.5,3.0,5.8,2.2,Iris-virginica
```

选择其中最受欢迎的 Iris 数据集，在网站 http://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data 上下载它，存放在自己的电脑上，命名为 Iris.txt

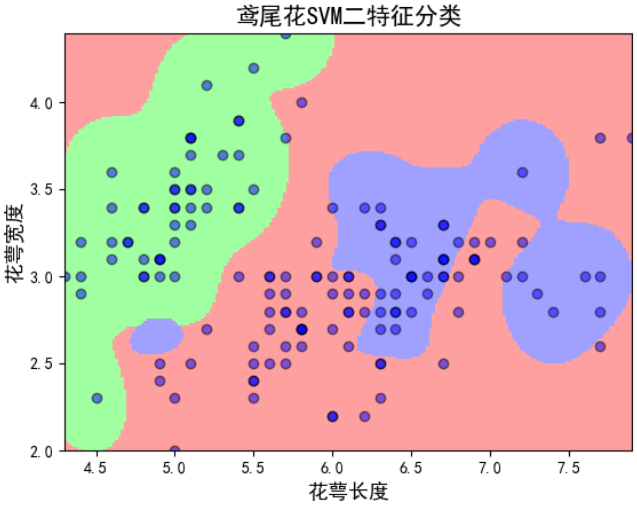
30.鸢尾花的分类预测.py

```
# -*- coding:utf-8 -*-
from sklearn import svm
import numpy as np
import matplotlib.pyplot as plt
import matplotlib as mpl
from matplotlib import colors
from sklearn.model_selection import train_test_split
"""  当使用 numpy 中的 loadtxt 函数导入该数据集时，假设数据类型 dtype 为浮点型，但是很明显第五列的数据类型并不是浮点型。
    因此我们要额外做一个工作，即通过 loadtxt()函数中的 converters 参数将第五列通过转换函数映射成浮点类型的数据。
    首先，我们要写出一个转换函数:    """
def iris_type(s):
    it = {'Iris-setosa': 0, 'Iris-versicolor': 1, 'Iris-virginica': 2}
    return it[s]
path = 'Iris.txt'                                # 数据文件路径
"""     converters：将数据列与转换函数进行映射的字典。eg：{1:fun}，含义是将第 2 列对应转换函数进行转换
    接下来读入数据，converters={4: iris_type}中“4”指的是第 5 列："""
data = np.loadtxt(path, dtype=float, delimiter=',', converters={4: iris_type})
""" 1. split(数据，分割位置，轴=1（水平分割） or 0（垂直分割））。
    2. x = x[:, :2]是为方便后期画图更直观，故只取了前两列特征值向量训练。
    3. sklearn.model_selection.train_test_split 随机划分训练集与测试集。
    train_test_split(train_data,train_target,test_size=数字, random_state=0)
    参数解释：
    train_data：所要划分的样本特征集
    train_target：所要划分的样本结果
    test_size：样本占比，如果是整数的话就是样本的数量
    random_state：是随机数的种子。     """
x, y = np.split(data, (4,), axis=1)
x = x[:, :2]
x_train, x_test, y_train, y_test = train_test_split(x, y, random_state=1, train_size=0.6)
"""    kernel='linear'时，为线性核，C 越大分类效果越好，但有可能会过拟合（defaul C=1）。
     kernel='rbf'时（default），为高斯核，gamma 值越小，分类界面越连续；gamma 值越大，分类界面
```

```
越“散”，分类效果越好，但有可能会过拟合。
    decision_function_shape='ovr'时，为one v rest，即一个类别与其他类别进行划分，
    decision_function_shape='ovo'时，为one v one，即将类别两两之间进行划分，用二分类的方法模拟多分类的结果 """
clf = svm.SVC(C=0.8, kernel='rbf', gamma=20, decision_function_shape='ovr')
clf.fit(x_train, y_train.ravel())
x1_min, x1_max = x[:, 0].min(), x[:, 0].max()  # 第0列的范围
x2_min, x2_max = x[:, 1].min(), x[:, 1].max()  # 第1列的范围
x1, x2 = np.mgrid[x1_min:x1_max:200j, x2_min:x2_max:200j] # 生成网格采样点
print x1_max, x1_min, x2_max, x2_min
grid_test = np.stack((x1.flat, x2.flat), axis=1)  # 测试点
print grid_test
mpl.rcParams['font.sans-serif'] = [u'SimHei']     # 指定默认字体
mpl.rcParams['axes.unicode_minus'] = False
cm_light = mpl.colors.ListedColormap(['#A0FFA0', '#FFA0A0', '#A0A0FF'])
cm_dark = mpl.colors.ListedColormap(['g', 'r', 'b'])
"""    pcolormesh(x,y,z,cmap)这里参数代入x1，x2，grid_hat，cmap=cm_light绘制的是背景。
     scatter中edgecolors是指描绘点的边缘色彩，s指描绘点的大小，cmap指点的颜色。
     xlim指图的边界    """
grid_hat = clf.predict(grid_test)                # 预测分类值
print grid_hat
grid_hat = grid_hat.reshape(x1.shape)                    # 使之与输入的形状相同
alpha = 0.5
plt.pcolormesh(x1, x2, grid_hat, cmap=cm_light)          # 预测值的显示
plt.plot(x[:, 0], x[:, 1], 'o', alpha=alpha, color='blue', markeredgecolor='k')
plt.scatter(x_test[:, 0], x_test[:, 1], s=120, facecolors='none', zorder=10)
                                                                  # 圈中测试集样本
plt.xlabel(u'花萼长度', fontsize=13)
plt.ylabel(u'花萼宽度', fontsize=13)
plt.xlim(x1_min, x1_max)
plt.ylim(x2_min, x2_max)
plt.title(u'鸢尾花SVM二特征分类', fontsize=15)
plt.show()
```

运行结果如下：

```
7.9 4.3 4.4 2.0
[[ 4.3        2.        ]
 [ 4.3        2.0120603]
 [ 4.3        2.0241206]
 ...,
 [ 7.9        4.3758794]
 [ 7.9        4.3879397]
 [ 7.9        4.4      ]]
[ 1.  1.  1. ...,  1.  1.  1.]
```



支持向量机(support Vector machine,SVM)是一种高级分类方法,它的任务是找到一个分隔两个数据集的最大边缘超平面,即这个分界线要尽量远离两边,当遇到普通的线性不可分时,采用核函数将原始数据映射到高维空间,在高维空间中就能实现线性分离了。但是实际情况往往不能分得非常清楚，这时可以考虑软边缘，它破例允许个别样本跑到其他类别中去，需要用到对错误分类的惩罚程度(软边缘参数)C

机器学习中始终要避免**过拟合**(overfitting)某个训练数据而在新的测试集上泛化性能差的问题。过拟合即学习进行得太彻底以至于学到了过多的局部特征和噪声带来的错误特征。这里的特征选择就相当于考试划重点。比如说能够在天空中那些随机分布的星星中连出星座的形状来，这是一阶过拟合，星座又和人的性格扯上关系，这是二阶过拟合。

```
In [2]: from sympy import *

In [3]: init_session()

In [6]: diff((x**3+3*x)/E**x,x)
Out[6]:
```

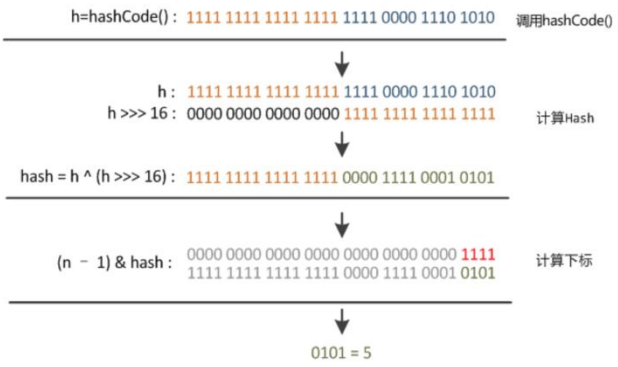
求导

$$(3x^2+3)e^{-x}-(x^3+3x)e^{-x}$$

```
In [10]: float(log(10))
Out[10]:
```

求ln2

2.30258509299




```
In [11]: integrate(log(x+1)/(2-x)**2,(x,0,1))
Out[11]:
```

定积分

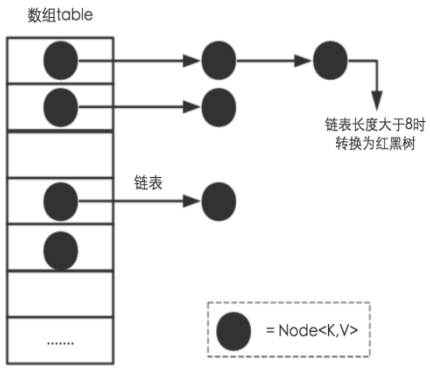
log(2)/3

并发的多线程场景下使用 HashMap 可能发生死锁。应换成线程安全的 ConcurrentHashMap

```
In [12]: summation(1/2**n,(n,1,oo))
Out[12]:
```

求和(无穷级数)

1




```
In [13]: expand((2*x+1)*(3*y+2))
Out[13]:
```

多项式展开

$6xy+4x+3y+2$

```
In [14]: N(factorial(100))
Out[14]:
```

科学计数法

$9.33262154439442\cdot 10^{157}$

```
In [15]: x,y,A,B,C,D=symbols('x y A B C D')

In [16]: solve(A*x*x+B*x+C-(D+y),x)
Out[16]:
```

解方程

$$\left[\left(-B+\sqrt{-4AC+4AD+4Ay+B^2}\right)/2A,\quad -\left(B+\sqrt{-4AC+4AD+4Ay+B^2}\right)/2A\right]$$

```
In [17]: solve((x*4+y-2-A,x+2-B), x,y)
Out[17]:
```

$\{x:B-2,\quad y:A-4B+10\}$

5. 和为k的连续子数组.java

给定一个整数数组和一个整数 k，你需要找到总和等于 k 的连续子数组的总数。

想法：使用 HashMap 存储键值对<累计总和,累计总和的重复出现次数>

```
public int subarraySum(int[] nums, int k) {
    int res=0,sum=0;
    HashMap<Integer,Integer> presum=new HashMap<>();
    presum.put(0,1);
    for(int i=0;i<nums.length;i++){
        sum+=nums[i];
```

```
            if(presum.containsKey(sum-k))
                res+=presum.get(sum-k);
            presum.put(sum,presum.getOrDefault(sum,0)+1);
        }
        return res;
    }
```

65.独特的数.C

一个 n 个元素的数组中除了一个独特的数只出现过一次外，其他的数都出现了两次，找到并输出这个独特的数，要求时间复杂度为 O(N)，且不开辟新的内存空间。

可以用位运算，首先来看看异或运算的性质：

```
    int a=2,b=3;
    printf("%d\n",a^0);          //2
    printf("%d\n",a^a);          //0
    printf("%d\n",a^b^a);        //3
```

利用这个性质，就能找到这个数

```
#include<stdio.h>
int singleNumber(int A[],int n)
{
    int result=0;
    for(int i=0;i<n;i++)
        result^=A[i];
    return result;
}
int main()
{
    int d[]={4,3,5,8,6,9,5,6,2,4,8,9,3},n=sizeof(d)/sizeof(d[0]);
    printf("%d\n",singleNumber(d,n));       //2
    return 0;
}
```

19.圆桌上的幸存者.Cpp

圆桌上有 1 到 1000 号，1 号右手边是 2 号，左手边是 1000 号。1 号开枪打死 2 号，把枪交给 3 号，3 号打死 4 号交给 5 号。。999 号打死 1000 号后把枪交给 1 号，继续循环，最后留下来的是几号？

使用 STL 暴力破解：

```
#include<iostream>
#include<list>
int main()
{
    std::list<int> people;
    for(int i=0;i<1000;i++)
        people.push_back(i+1);
    while(people.size()>1)
    {
        int criminal=people.front();
        people.pop_front();          //在这中间加一句输出队首元素
        people.pop_front();          //可以打印出完整的枪毙名单
        people.push_back(criminal);
    }
    std::cout<<"the winner is "<<people.back()<<std::endl;
    return 0;
}
```

输出：

```
the winner is 977
```

31.两个杯子.py

有一种玻璃杯质量确定但未知，需要检测，有一栋 200 层的大楼，现在给你两个这样的一模一样的杯子，如果从 n 层扔下杯子，杯子不碎，那么从前 n-1 层扔下去杯子都不会碎。提出一个策略，要保证能测出杯子从哪个楼层扔下来恰好会碎，并且在最坏情况下所扔次数最少

很明显，这种情况下二分法是不适用的。为了保证测试次数最少，每次扔的区间减少一层，这样可以保证每个区间查找的最坏情况下的次数是一样的。比如一开始在 14 楼开始，没碎的话再在 27 楼扔，然后是 39 楼。。。一旦碎掉，就在上一次没有碎的楼层开始逐层往上测试。

实际上这是一个动态规划问题，设 f(n,m)为 n 层楼、m 个杯子的最少测试次数，那么

f(0,m)=0　　m>=1

f(n,1)=n　　n>=1

f(n,m)=$\min\limits_{1 \le i \le n}$\{max\{f(i-1, m-1), f(n-i, m)\}\} + 1

i 表示这一次测试是把杯子从 i 楼扔下去，那么就有两种结果，如果杯子碎了，那么剩余杯子数目是 m-1，接下来的楼层测试范围只能是 0 到 i-1，接下来的最少测试次数就是 f(i-1, m-1)；如果杯子没碎，剩余杯子数目还是 m，剩余的测试范围是 i 到 n，即 n-i。题目意思是“最坏情况下的最少次数”，为了得到最坏情况，就要取这两种情况中次数更多的那个，为了得到最少次数，就要对从 0 到 n 的所有 i 取最小值。

给出 python3 的代码：

```
import functools
@functools.lru_cache(maxsize=None)
def f(n, m):
    if n == 0:
        return 0
    if m == 1:
        return n
    ans = min([max([f(i-1, m-1), f(n-i, m)])for i in range(1, n+1)])+1
    return ans
print(f(100, 2))          #14
print(f(200, 2))          #20
```

32.手机的解锁图案.py

手机的九宫格解锁需要满足的要求是：至少经过 4 个点；不能重复经过同一个点；路径上的中间点不能跳过(如从 1 到 3 一定会经过 2)；如果中间的点是之前用过的，那么这个点就可以被跳过(如 2..13，因为 2 已经被用过，1 就可以越过 2 与 3 连接，在原本的 2..123 中省略了 2)

现在问你总共有多少种可能的解锁图案

先计算出所有大于 4 个数字的所有排列组合，然后从中去掉穿过中间那个数字的组合，剩下的就是符合要求的

```
from itertools import chain, permutations
impossible = {'13': '2', '46': '5',
             '79': '8', '17': '4',
             '28': '5', '39': '6',
             '19': '5', '37': '5',
             '31': '2', '64': '5',
             '97': '8', '71': '4',
             '82': '5', '93': '6',
             '91': '5', '73': '5'}
def counts():
```

```
    iterlst = chain(*(permutations('123456789', i) for i in range(4, 10)))
    count = 0
    for i in iterlst:
        stri = ''.join(i)
        for k, v in impossible.items():
            if k in stri and v not in stri[:stri.find(k)]:
                break
        else:
            count += 1
    return count
print (counts())            #389112
```

33.可怜的猪.py

1000 桶水，其中一桶有毒，猪喝毒水后会在 15 分钟内死去，想用 1 个小时的时间找到这桶毒水，至少需要几头猪。假设猪喝一桶水花费的时间可以忽略

采用信息论的观点思考这个问题，1 头猪在 1 个小时内只能提供 5 种信息：

0-15 分钟死/15-30 分钟死/30-45 分钟死/45-60 分钟死/不死

那么 n 头猪能够提供的信息量是 5 的 n 次方。而我们在具体测试的时候能够建立一种信息的对应关系(具体测试的做法并不重要)：猪在哪个时间段死对应了哪桶水是有毒的，即两者的信息量是相同的(这才是关键)。确定 1000 桶水中哪一桶是毒水，需要的信息量是 1000，那么，猪提供的信息量必须超过 1000 才能找到毒水。

```
def poorPigs(buckets,minutesToDie,minutesToTest):
    pigs = 0
    while (minutesToTest / minutesToDie + 1) ** pigs < buckets:
        pigs += 1
    return pigs
print (poorPigs(1000, 15, 60))      #5
```

编程语言的 4 种类型：

强类型：偏向于不容忍隐式类型转换

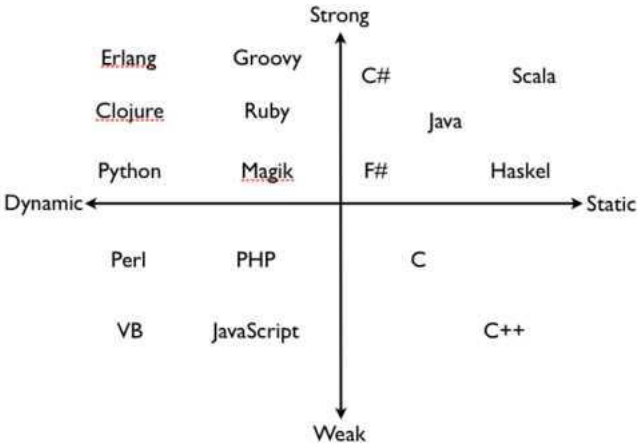
如 "1"+2 会报错

弱类型：偏向于容忍隐式类型转换

如 "1"+2='12'

静态类型：编译(或解释)时就知道每一个变量的类型

动态类型：编译(或解释)时不知道每一个变量的类型



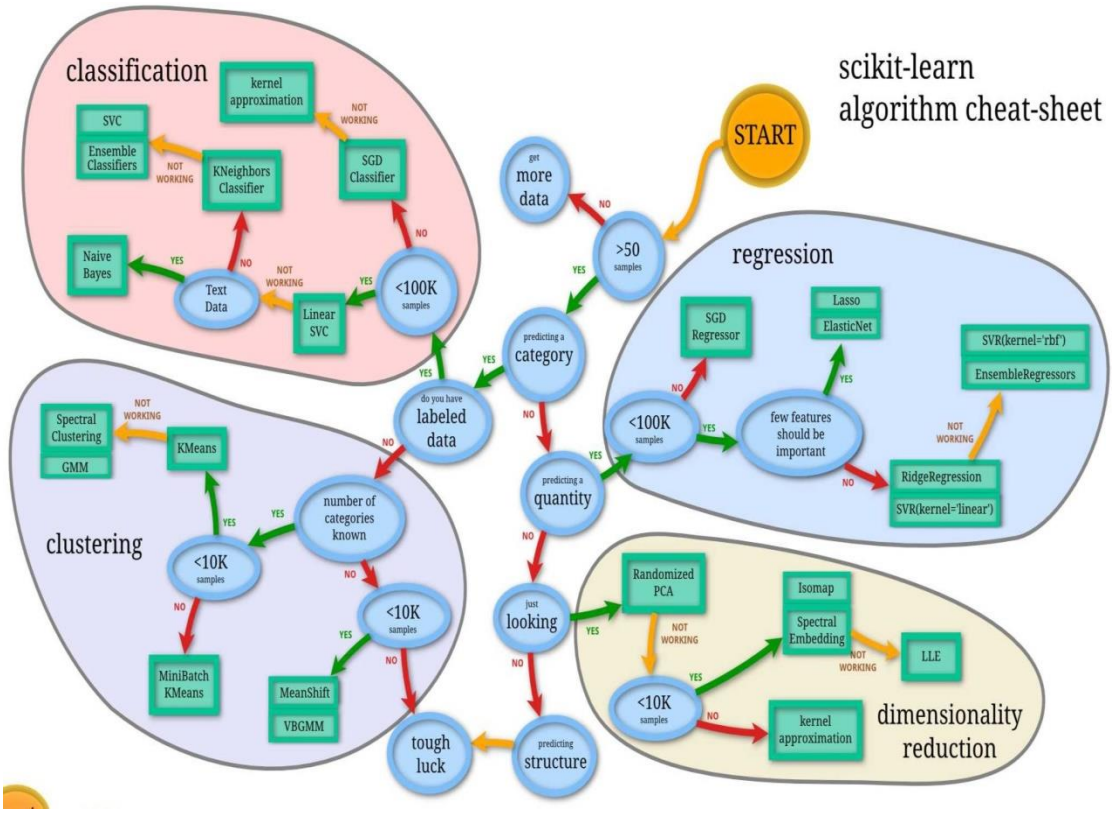
这里有两个有意思的 Linux 命令：

rm –rf/　　删除根目录下的全部文件。效果：整台设备变成砖头

:(){:|:&};:　　用冒号为名称声明一个函数，功能是用新的 shell 打开一个后台任务，任务是用一个管道连接到自身的两次调用，然后开始执行函数。效果：电脑卡爆

给定一个 m×n 矩阵，如果一个元素为 0，则将其整个行和列设置为 0。

我的想法很简单：将每行的状态存储在该行的第一行中，并将每列的状态存储在该列的第一行中。 由于 row0 的状态和 column0 的状态将占据同一个单元格，因此我将它作为 row0 的状态，并将另一个变量“col0”用于 column0。 在第一阶段，使用矩阵元素以自上而下的方式设置状态。 在第二阶段，使用状态以自底向上的方式设置矩阵元素。



34.利用身高、体重、性别预测胖瘦.py

```
train_data = [[160, 60, 1], [155, 80, 1], [178, 53, 2], [158, 53, 2], [166, 45, 2],
              [170, 50, 2], [156, 56, 2], [166, 50, 1], [175, 55, 1], [188, 68, 1],
              [159, 41, 1], [166, 70, 1], [175, 85, 1], [188, 98, 1], [159, 61, 2]]
train_target = [1, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 1, 1, 1]
test_data = [[166, 45, 2], [172, 52, 1], [156, 60, 1], [150, 70, 2]]
test_target = [0, 0, 1, 1]
from sklearn import svm
clf = svm.SVC()
clf.fit(train_data, train_target)
result = clf.predict(test_data)
print(result.tolist())
from sklearn.naive_bayes import GaussianNB
gnb = GaussianNB().fit(train_data, train_target)
result = gnb.predict(test_data)
print(result)
from sklearn.svm import LinearSVC
clf = LinearSVC()
clf.fit(train_data, train_target)
result = clf.predict(test_data)
print(result)
from sklearn.linear_model import LogisticRegression
clf = LogisticRegression().fit(train_data, train_target)
result = clf.predict_proba(test_data)
print(result)
```

输出结果：

```
[0, 1, 1, 1]
[0 0 1 1]
[0 0 0 1]
[[ 0.95098932  0.04901068]
 [ 0.85844258  0.14155742]
 [ 0.18885402  0.81114598]
 [ 0.01246901  0.98753099]]
```

有两个长度分别为 m 和 n 的已经排好序的数组，找到并输出它们的整体的中位数，要求算法的时间复杂度为 O(log(min(m,n)))

```
    double findMedianSortedArrays(vector<int>& nums1, vector<int>& nums2)
    {
        int N1 = nums1.size();
        int N2 = nums2.size();
        if(N1 < N2)
            return findMedianSortedArrays(nums2, nums1);
        int lo = 0, hi = N2 * 2;
        while(lo <= hi){
            int mid2 = (lo + hi) / 2;
            int mid1 = N1 + N2 - mid2;
            double L1 = (mid1 == 0) ? INT_MIN : nums1[(mid1-1)/2];
            double L2 = (mid2 == 0) ? INT_MIN : nums2[(mid2-1)/2];
            double R1 = (mid1 == N1*2) ? INT_MAX : nums1[mid1/2];
            double R2 = (mid2 == N2*2) ? INT_MAX : nums2[mid2/2];
            if(L1 > R2)
                lo = mid2 + 1;
            else if(L2 > R1)
                hi = mid2 - 1;
            else
                return (max(L1, L2) + min(R1, R2)) / 2;
        }
        return -1;
    }
```

算法的原理是对两个数组分别进行二分查找，每次去掉两个数组总共 1/2 的数据

7.数组的最长连续序列.java

给定一个未排序的整数数组，求它的最长连续元素子序列的长度，例如给定 [29,5,7,100,6,20,8]，最长连续序列为[5,6,7,8]，长度为 4。要求算法的时间复杂度为 O(N)

很少有人想得到 java 中的 HashMap，苛刻的复杂度需要特殊的数据结构(最好有哈希特性)，

We will use HashMap. The key thing is to keep track of the sequence length and store that in the boundary points of the sequence. For example, as a result, for sequence {1, 2, 3, 4, 5}, map.get(1) and map.get(5) should both return 5.

Whenever a new element n is inserted into the map, do two things:

1. See if n − 1 and n + 1 exist in the map, and if so, it means there is an existing sequence next to n. Variables left and right will be the length of those two sequences, while 0 means there is no sequence and n will be the boundary point later. Store (left + right + 1) as the associated value to key n into the map.
2. Use left and right to locate the other end of the sequences to the left and right of n respectively, and replace the value with the new length.

Everything inside the for loop is O(1) so the total time is O(n).

```
import java.util.*;
public class Helloworld {
    public static int longestConsecutive(int[] num){
        int res=0;
        HashMap<Integer,Integer> map=new HashMap<Integer,Integer>();
        for(int n:num){
            if(!map.containsKey(n)){
                int left=(map.containsKey(n-1))?map.get(n-1):0;
                int right=(map.containsKey(n+1))?map.get(n+1):0;
                int sum=left+right+1;
                map.put(n,sum);
                res=Math.max(res,sum);
```

```
                map.put(n-left,sum);
                map.put(n+right,sum);
            }else
                continue;
        }
        return res;
    }
    public static void main(String[] args){
        int[] a={100,4,200,1,3,2};
        int result=longestConsecutive(a);
        System.out.println(result);    //4
    }
}
```

8.最长有效圆括号.java

给你一个只包含'(' 和')'的字符串，求出其中最长的有效的子串的长度，要求时间复杂度为O(N)。例如")()())"中的最长有效子串是"()()"，长度为4

```
import java.util.*;
public class Helloworld {
    public static int longestValidParentheses(String s){
        Stack<Integer> stack=new Stack<Integer>();
        int res=0;
        int left=-1;
        for(int j=0;j<s.length();j++){
            if(s.charAt(j)=='(')
                stack.push(j);
            else{
                if(stack.isEmpty())
                    left=j;
                else{
                    stack.pop();
                    if(stack.isEmpty())
                        res=Math.max(res, j-left);
                    else
                        res=Math.max(res, j-stack.peek());
                }
            }
        }
        return res;
    }
    public static void main(String[] args){
        String a="())()()(())()()()()()(()))(())(((";
        int b=longestValidParentheses(a);
        System.out.println(b);           //12
    }
}
```

给定一个只包含正整数的非空数组，查找数组是否可以分割成两个子集，使得两个子集的元素总和相等。

```
#include<iostream>
#include<vector>
#include<algorithm>
#include<bitset>
using namespace std;
bool canPartition(vector<int>& nums) {
    int sum=accumulate(nums.begin(),nums.end(),0);
    bitset<10001> bits(1);
    for(auto n:nums)
        bits|=bits<<n;
    return !(sum%2)&&bits[sum/2];
}
int main(void){
    vector<int> a{1,5,5,11};
```

```
    cout<<canPartition(a);        //1
}
```

bits 的第 i 位为 1 的话表示此数组里面存在组合使得该组合的和为 i。

此处采用归纳法简单的分析下算法：

1.假设 n 之前的子列里面存在 1~m,k~L……之间和的组合

2.填加了数字 n 之后，将会存在(1+n)~(m+n),(k+n)~(L+n)……之间和的组合（只要在上面的组合里面添加当前的元素 n 即可），在标记 bits 里面相当于将 bits 向左边移动 n 位，即 bits << n

3.故目前为止，存在 1~m,k~L……以及(1+n)~(m+n),(k+n)~(L+n)……之间和的组合

4.故 bits |= bits << n

给定一个未排序的数组 nums，重新排序使得 nums [0] <nums [1]> nums [2] <nums [3] ...

```
#include<iostream>
#include<algorithm>
#include<vector>
using namespace std;
void wiggleSort(vector<int> &nums){
    int n=nums.size();
    auto midptr=nums.begin()+n/2;
    nth_element(nums.begin(),midptr,nums.end());    //find a median
    int mid=*midptr;
    #define A(i) nums[(1+2*(i))%(n|1)]
                                    //A(0~9) means nums[1,3,5,7,9,0,2,4,6,8]
    int i=0,j=0,k=n-1;
    while(j<=k){  //3-way-partition-to-wiggly in O(n) time with O(1) space.
        if(A(j)>mid)
            swap(A(i++),A(j++));
        else if(A(j)<mid)
            swap(A(j),A(k--));
        else
            j++;
    }
}
int main(void){
    vector<int> a{1,2,3,4,5,6,7,8,9};
    wiggleSort(a);
    for(int i:a)
        cout<<i<<' ';       //5 9 3 7 2 6 4 8 1
    return 0;
}
```

Java 面试题：

int 和 Integer 哪个会占用更多的内存？

Integer 对象会占用更多的内存。Integer 是一个对象，需要存储对象的元数据。但是 int 是一个原始类型的数据

解释内存中的栈(stack)、堆(heap)和静态区(static area)的用法？

通常我们定义一个基本数据类型的变量，一个对象的引用，还有就是函数调用的现场保存都使用内存中的栈空间；而通过 new 关键字和构造器创建的对象放在堆空间；程序中的字面量（literal）如直接书写的 100、”hello”和常量都是放在静态区中。栈空间操作起来最快但是栈很小，通常大量的对象都是放在堆空间，理论上整个内存没有被其他进程使用的空间甚至硬盘上的虚拟内存都可以被当成堆空间来使用

“a==b”和”a.equals(b)”有什么区别？

如果 a 和 b 都是对象，则 a==b 是比较两个对象的引用，只有当 a 和 b 指向的是堆中的同一个对象才会返回 true，而 a.equals(b) 是进行逻辑比较，所以通常需要重写该方法来提供逻辑一致性的比较。例如，String 类重写 equals() 方法，所以可以用于两个不同对象，但是包含的字母相同的比较。

a.hashCode() 有什么用？与 a.equals(b) 有什么关系？

hashCode() 方法是相应对象整型的 hash 值。它常用于基于 hash 的集合类，如 Hashtable、HashMap、LinkedHashMap 等等。它与 equals() 方法关系特别紧密。根据 Java 规范，两个使用 equal() 方法来判断相等的对象，必须具有相同的 hash code

final、finalize 和 finally 的不同之处？

final 是一个修饰符，可以修饰变量、方法和类。如果 final 修饰变量，意味着该变量的值在初始化后不能被改变。Java 技术允许使用 finalize() 方法在垃圾收集器将对象从内存中清除出去之前做必要的清理工作。这个方法是由垃圾收集器在确定这个对象没有被引用时对这个对象调用的，但是什么时候调用 finalize 没有保证。finally 是一个关键字，与 try 和 catch 一起用于异常的处理。finally 块一定会被执行，无论在 try 块中是否有发生异常

ArrayList 与 LinkedList 的区别？

最明显的区别是 ArrrayList 底层的数据结构是数组，支持随机访问，而 LinkedList 的底层数据结构是双向链表，不支持随机访问。使用下标访问一个元素，ArrayList 的时间复杂度是 O(1)，而 LinkedList 是 O(n)

String 是最基本的数据类型吗？

不是。Java 中的基本数据类型只有 8 个：byte、short、int、long、float、double、char、boolean；除了基本类型（primitive type），剩下的都是引用类型（reference type），枚举类型（enumeration type）也是引用类型

访问修饰符 public,private,protected,以及不写（默认）时的区别？

类的成员不写访问修饰时默认为 default。默认对于同一个包中的其他类相当于公开（public），对于不是同一个包中的其他类相当于私有（private）。受保护（protected）对子类相当于公开，对不是同一包中的没有父子关系的类相当于私有。Java 中，外部类的修饰符只能是 public 或默认，类的成员（包括内部类）的修饰符可以是以上四种

StringBuffer，StringBuilder 区别是啥？

这是个字符串内存管理策略问题。几十年前 在 C 和 Pascal 的时代，就有字符串存储形式 是 \0 结尾还是 长度+内容的争论，这个问题严重影响了 API 的设计，内存的管理，甚至程序架构。字符串是一种非常常用的 生命周期通常很短的对象，而且它的 size 是不固定的，长度为 1 或 1GB 都有可能，导致内存管理非常麻烦复杂。用长度+内容表示，每个串都要额外付出 4 个字节，用\0 结尾吧，算长度就很麻烦，万一结尾忘了\0 就要崩溃，选哪个好字符串的长度放哪里，放到起始指针的位置，还是起始指针的前面如果放前面，那么字符串起始指针和内存块起始不一致怎么解决字符串拼接的时候把源串复制到目标串结尾，那么目标串剩余内存不够怎么办，重新分配要多一次赋值，频繁拼接性能有问题怎么办函数返回值如果是字符串，那么这个串是调用者分配内存还是被调用者分配，谁来负责释放。如果调用者分配，那么调用者怎么知道字符串将有多长。频繁分配释放大小各异的字符串，会不会导致内存碎片化对于小字符串是分配在堆上还是栈上怎么把常量串和变量串分别处理如果设计面向对象的字符串，字符串加法怎么定义，是在原来对象上加，还是生成一个新对象如果每加一次都生成新对象会不会导致构造析构太频繁如果是托管语言会不会太频繁 GC 要不要设计单独的辅助类来解决字符串拼接问题那这个辅助类怎么设计，要不要考虑线程安全如果考虑线程安全的话，怎么兼顾性能你把这些问题都思考透彻了，再遇到问此类问题的面试官，就给他上上课，保证他懵逼

	String	***StringBuffer***	***StringBuilder***
Storage Area	Constant String Pool	Heap	Heap
Modifiable	No (immutable)	Yes(mutable)	Yes(mutable)
Thread Safe	Yes	Yes	No
Performance	Fast	Very slow	Fast

HashMap 为什么要扩容?

hashmap 进行扩容主要为了减少冲突率，提高查询的性能。loadFactor=size/capacity 。loadFactor 越大冲突率越高

9.Helloworld 程序(版本 3).java

```
    public static void main(String[] args){
        if(System.out.printf("hello")==null)
            System.out.print("hello");
        else
            System.out.print(" world");
    }
```

66.n 个数中前 m 大的数.C

在 n 个互不相同的位于区间[-500000,500000]的整数中找到前 m 大的数并按从大到小的顺序输出

```
#include<stdio.h>
#define OFFSET 500000              //偏移量，用于补偿实际数字与数组下标之间的偏移
int Hash[OFFSET*2+1];              //Hash 数组：记录每个数字是否出现，
int main()
{
    int n,m;
    while(scanf("%d%d",&n,&m)!=EOF)
    {
        for(int i=-OFFSET;i<=OFFSET;i++)
            Hash[i+OFFSET]=0;               //初始化，将每个数字标记为未出现
        for(int i=1;i<=n;i++)
        {
            int x;
            scanf("%d",&x);
            Hash[x+OFFSET]=1;           //凡是出现过的数字，该数组元素均被设置为 1
        }
        for(int i=OFFSET;i>=-OFFSET;i--)
        {                                           //输出前 m 个数字
            if(Hash[i+OFFSET]==1)
            {                                   //若该数字在输入中出现
                printf("%d",i);                     //输出它
                m--;                        //每输出一个数字，m 减一，直到变为 0
                if(m!=0)
                    printf(" ");                        //注意格式
                else
                {                                        //输出完毕
                    printf("\n");                       //最后换行
                    break;                              //跳出循环
                }}}}
    return 0;
}
```

弗洛伊德判圈算法：判断链表中是否存在环，如果存在，返回环的起始结点

首先创建两个快慢指针 1 和 2，同时让它们指向链表的头结点，然后开始主循环。在每一次循环中，让指针 1 移动 1 个结点，让指针 2 移动 2 个结点。然后判断此时两个指针指向的结点是否相同，如果相同，则链表中存在环；如果不相同，继续下一次循环。

很简单的道理，假设在一个环形跑道上，两个运动员同时起跑，如果一个快一个慢，跑得快的那个总会再一次超过跑得慢的，因为跑道是环形的。

快慢指针相遇之后，把其中一个指针移到链表头部，然后同时一起以同样的速度移动，再次相遇的地点就是环的起始结点。

20.最小窗口子串.Cpp

给出一个字符串 S 和一个字符串 T，找到 S 中的最小窗口，它将包含复杂度为 O（n）的 T 中的所有字符。例如，S =“ADOBECODEBANC”　　T =“ABC”　　最小的窗口是“BANC”。

注意：

如果 S 中没有覆盖 T 中所有字符的窗口，则返回空字符串“ ”。

如果有多个这样的窗口，则保证在 S 中总是只有一个唯一的最小窗口。

对于大多数子字符串问题，我们给了一个字符串，并且需要找到满足一些限制的子字符串。一般的方法是使用两个指针辅助的 hashmap。

有一点需要提到的是，当要求查找最大子字符串时，我们应该在内部 while 循环后更新最大值，以保证子字符串是有效的。 另一方面，当要求查找最小子字符串时，我们应该更新内部 while 循环内的最小值。

这是处理子串问题的通用模板：

```
int findSubstring(string s){
        vector<int> map(128,0);
        int counter;                    // check whether the substring is valid
        int begin=0, end=0;             //two pointers, one point to tail and one   head
        int d;                          //the length of substring
        for() { /* initialize the hash map here */ }
        while(end<s.size()){
            if(map[s[end++]]-- ?){   /* modify counter here */ }
            while(/* counter condition */){
                 /* update d here if finding minimum*/
               //increase begin to make it invalid/valid again
               if(map[s[begin++]]++ ?){ /*modify counter here*/ }
            }
            /* update d here if finding maximum*/
        }
        return d;
  }
```

```
#include<iostream>
#include<string>
#include<map>
#include<vector>
using namespace std;
string minWindow(string s, string t)
{
    vector<int> map(128, 0);
    for (auto c : t)
        map[c]++;
    int counter = t.size(), begin = 0, end = 0, d = INT_MAX, head = 0;
    while (end < s.size())
    {
        if (map[s[end++]]-- > 0)
            counter--;
        while (counter==0)
        {
            if (end - begin < d)
                d = end - (head = begin);
            if (map[s[begin++]]++ == 0)
                counter++;
        }
    }
    return d == INT_MAX ? "" : s.substr(head, d);
}
int main(void)
{
    string a = "ADOBECODEBANC", b = "ABC";
    cout<<minWindow(a, b)<<endl;
}
```

21.直方图中的最大矩形.Cpp

给定 n 个非负整数，表示每个小节的宽度为 1 的直方图的小节高度，找到直方图中最大矩形的面积。例如，给定高度= [2,1,5,6,2,3]，返回 10

我将一个哨兵节点推回到高度的末端，以使代码逻辑更加简洁。

```
#include<iostream>
#include<vector>
using namespace std;
int LargestRectangleArea(vector<int> &height){
    int ret=0;
    height.push_back(0);
    vector<int> index;
    for(int i=0;i<height.size();i++){
        while(index.size()>0 && height[index.back()]>=height[i]){
            int h=height[index.back()];
            index.pop_back();
            int sidx=index.size()>0 ? index.back() : -1;
            if(h*(i-sidx-1)>ret)
                ret=h*(i-sidx-1);
        }
        index.push_back(i);
    }
    return ret;
}
int main(void){
    int a[]={2,1,5,6,2,3},n=sizeof(a)/sizeof(a[0]);
    vector<int> A;
    for(int i=0;i<n;i++)
        A.push_back(a[i]);
    cout<<LargestRectangleArea(A);  //10
}
```

35.二维数据的示例操作.py

```
#coding=utf-8
import pandas as pd
import numpy as np
a = np.ones((3, 5))        #生成一个全为 1 的 3 行 5 列的矩阵
a = np.eye(3)      #生成一个 3 乘 3 的单位矩阵
print(a.nonzero())      #返回由两个 array 组成的元组，分别为按行优先顺序遍历矩阵 a 时 a 中非零元素的行、列索引值
print(a[a.nonzero()])      #返回矩阵 a 中非零元素组成的一维矩阵
np.random.seed(1)
b = np.random.randint(0, 10, size=(6, 5))
print(b[:, 1])         #选取一列，逗号分隔，行在前，列在后
print(np.mean(b))      #求全部元素的平均数
print(sum(b))          #对每一列求和
print(sum(sum(b)))              #对全部元素求和
print(np.cumsum(b)) #对全部元素按行优先方式累计求和。参数 axis=1 和 0 分别表示按行和列累计求和，返回同样大小的矩阵
b = pd.DataFrame(b, index=list('abcdef'), columns=list('opqrs'))
print(b.sum(axis=1))                #对每一行求和
print(b[['p', 'q']])           #选取两列
print(b[:2])                  #行切片，右边开区间
print(b.ix[1])                #选取一行，返回 Series 对象
print(b.ix[['c', 'e'], ['p', 'q']])     #同时选取行和列，行在前，列在后
print(b.ix[[3, 4], [2]])       #也可以按数字顺序索引
print(b.ix[:, [2]])            #取出第 3 列
print(b.describe()) #对 Series 或 DataFrame 的每一列返回非 Nan 值的数量、平均值、标准差、最大最小值、汇总统计。对非数值型数据会给出另一种汇总结果
print(b.sort_values(by='r'))       #按其中一列的值来排序，按值对 Series 排序可用 order() 方法
df = pd.DataFrame(np.random.randn(7, 3))
df.ix[:4, 1] = np.NaN; df.ix[:2, 2] = np.NaN; df[4] = np.NaN; df.ix[7] = np.NaN
print(df.isnull())              #判断每个元素位置是否缺失，返回一个布尔型 DataFrame
```

```
print(df.dropna(how='all'))            #去掉全为缺失值的行
print(df.dropna(how='all', axis=1))     #去掉全为缺失值的列
df = df.drop(4, axis=1)                #去掉索引值为 4 的列
df = df.drop(7)                        #去掉索引值为 7 的行
print(df.dropna())                     #去掉任何有缺失值的行
print(df.fillna(0))                    #所有缺失位置填充为 0
print(df.fillna({1: 0.5, 2: -1}))           #不同的列填充不同的值
print(df.fillna(method='bfill', limit=2))   #向上填充，最多填充 2 个，ffill 为向下填充
b.to_excel('practice.xlsx')                 #把 DataFrame 存入表格
```

```
In [1]: import numpy as np

In [2]: np.c_[1:6,-10:-5]
Out[2]:
array([[   1, -10],
       [   2,   -9],
       [   3,   -8],
       [   4,   -7],
       [   5,   -6]])

In [3]: a = np.arange(3)

In [4]: a.repeat(3)
Out[4]: array([0, 0, 0, 1, 1, 1, 2, 2, 2])

In [5]: a.repeat([2,3,4])
Out[5]: array([0, 0, 1, 1, 1, 2, 2, 2, 2])

In [6]: b=np.random.randn(2,2)

In [7]: b
Out[7]:
array([[-0.9874923 ,  0.13060323],
       [-1.15067078,  0.650308  ]])

In [8]: b.repeat(2,axis=0)
Out[8]:
array([[-0.9874923 ,  0.13060323],
       [-0.9874923 ,  0.13060323],
       [-1.15067078,  0.650308  ],
       [-1.15067078,  0.650308  ]])

In [9]: b.repeat([2,3],axis=1)
Out[9]:
array([[-0.9874923 , -0.9874923 ,  0.13060323,
0.13060323,  0.13060323],
       [-1.15067078, -1.15067078,
0.650308  ,  0.650308  ,  0.650308  ]])

In [10]: np.tile(b,2)
Out[10]:
array([[-0.9874923 ,  0.13060323, -0.9874923 ,
0.13060323],
       [-1.15067078,  0.650308  ,
-1.15067078,  0.650308  ]])

In [11]: np.tile(b,(3,2))
Out[11]:
array([[-0.9874923 ,  0.13060323, -0.9874923 ,
0.13060323],
       [-1.15067078,  0.650308  ,
-1.15067078,  0.650308  ],
       [-0.9874923 ,  0.13060323, -0.9874923 ,
0.13060323],
       [-1.15067078,  0.650308  ,
-1.15067078,  0.650308  ],
       [-0.9874923 ,  0.13060323, -0.9874923 ,
0.13060323],
       [-1.15067078,  0.650308  ,
-1.15067078,  0.650308  ]])

In [2]: a=np.arange(6).reshape((3,-1),order='F')

In [3]: b=np.random.randint(-3,0,size=(3,2))

In [5]: np.hstack((a,b))
Out[5]:
array([[ 0,  3, -1, -2],
       [ 1,  4, -2, -3],
       [ 2,  5, -2, -2]])

In [6]: np.vstack((a,b))
Out[6]:
array([[ 0,  3],
       [ 1,  4],
       [ 2,  5],
       [-1, -2],
       [-2, -3],
       [-2, -2]])

In [7]: c=np.arange(12).reshape(6,-1)

In [8]: first,second,third=np.split(c,[1,3])

In [9]: first
Out[9]: array([[0, 1]])

In [10]: second
Out[10]:
array([[2, 3],
       [4, 5]])

In [11]: third
Out[11]:
array([[ 6,  7],
       [ 8,  9],
       [10, 11]])
```

`np.where(a>0,1,a)` 将矩阵 a 中大于 0 的元素替换为 1

`a.sort()` 将矩阵 a 中每一行元素就地排序

`c.sort(0)` 将矩阵 c 中每一列元素就地排序

`np.unique(c)` 得到 c 中不重复元素的有序序列

np.in1d(c,[10,5,-9]) 得到一个表示 c 中元素是否包含于[10,5,-9]的布尔型数组
np.intersect1d(b,c) 计算 b 和 c 中的公共元素并返回有序结果
a=pd.DataFrame({'k1':['one']*3+['two']*4,'k2':[1,1,2,3,3,4,4]})
a.duplicated() 返回一个布尔型 Series，表示各行是否是重复行
a.drop_duplicates() 移除重复行，默认会判断所有列

22.陷阱雨水.Cpp

给定 n 个非负整数代表每个柱的宽度为 1 的高程图，计算下雨后可以捕集多少水。
例如，给定[0,1,0,2,1,0,1,3,2,1,2,1]，返回 6。

这是我的想法：不是按高度*宽度来计算面积，而是以累积的方式来考虑。 换句话说，每个箱子的总水量（宽度= 1）。
从左至右搜索，左右分别保持最大高度，就像部分容器的一面墙。 固定较高的一个，从下部流水。 例如，如果左边的当前高度较低，我们在左边的容器中加水。 直到左边遇见右边，我们填满整个容器。

```
int trap(int A[], int n){
    int left=0; int right=n-1;
    int res=0;
    int maxleft=0,maxright=0;
    while(left<=right){
        if(A[left]<=A[right]){
            if(A[left]>=maxleft)
                maxleft=A[left];
            else
                res += maxleft-A[left];
            left++;
        }else{
            if(A[right]>=maxright)
                maxright=A[right];
            else
                res += maxright-A[right];
            right--;
        }
    }
    return res;
}
#include<iostream>
int main(void){
    int A[]={0,1,0,2,1,0,1,3,2,1,2,1},n=sizeof(A)/sizeof(A[0]);
    std::cout<<trap(A,n);     //6
}
```

10.二叉树最大路径总和.java

给定一个二叉树，找到最大路径和。
对于这个问题，路径被定义为从父节点连接到树中任何节点的任意节点的任意序列。 路径必须包含至少一个节点，不需要经过根。
这是我的想法：

从开始到结束的路径，在树上上升 0 步或更多步，然后下降 0 步或更多步。 一旦下降，它不能上升。 每条路径都有一个最高的节点，也是路径上所有其他节点的最低共同祖

先。

递归方法 maxPathDown（TreeNode 节点）（1）计算最大路径和，其中最高节点是输入节点，必要时更新最大值（2）返回可以扩展到输入节点的父节点的路径的最大总和。

```
public class Treenode{
    int val;
    Treenode left;
    Treenode right;
    Treenode(int x){
        val=x;
    }
}
public class Helloworld {
    int maxValue;
    public int maxPathSum(Treenode root){
        maxValue=Integer.MIN_VALUE;
        maxPathDown(root);
        return maxValue;
    }
    private int maxPathDown(Treenode node){
        if(node==null)
            return 0;
        int left=Math.max(0, maxPathDown(node.left));
        int right=Math.max(0, maxPathDown(node.right));
        maxValue=Math.max(maxValue, left+right+node.val);
        return Math.max(left, right)+node.val;
    }
    public static void main(String[] args){}
}
```

23.合并 k 个排序链表.Cpp

合并 k 个已排序的链接列表并将其作为一个排序链表返回。 分析并描述其复杂度。

第一个函数是合并两个排序链表。这个解决方案不是一个尾递归，链表太长时堆栈会溢出，基本的想法非常简单。 我们可以合并前两个链表，然后推回去(push it back)。 继续这样做，直到只剩下一个链表。 实际上，我们可以把它看作是一个反复的分而治之的解决方案。

```
#include<vector>
using namespace std;
struct ListNode{
    int val;
    ListNode *next;
    ListNode(int x) : val(x), next(nullptr) {}
};
ListNode* mergeTwoLists(ListNode *l1,ListNode *l2){
    if(l1==nullptr){
        return l2;
    }
    if(l2==nullptr){
        return l1;
    }
    if(l1->val<=l2->val){
        l1->next=mergeTwoLists(l1->next,l2);
        return l1;
    }
    else{
        l2->next=mergeTwoLists(l1,l2->next);
```

```
        return l2;
    }
}
ListNode* mergeKLists(vector<ListNode *> &lists){
    if(lists.empty()){
        return nullptr;
    }
    while(lists.size()>1){
        lists.push_back(mergeTwoLists(lists[0],lists[1]));
        lists.erase(lists.begin());
        lists.erase(lists.begin());
    }
    return lists.front();
}
int main(void){}
```

24.二进制矩阵中的最大矩形.Cpp

给定一个填充了 0 和 1 的二进制矩阵，找到最大的只包含 1 的矩形并返回其面积。

DP 解决方案：从第一行开始逐行进行。 设第 i 行和第 j 列的最大矩形区域由

[right（i，j） - left（i，j）] * height（i，j）计算。

所有 3 个变量 left，right 和 height 都可以由前一行的信息来确定，也可以由当前行的信息来决定。 所以它可以被看作是一个 DP 解决方案。 转换方程是：

left（i，j）= max（left（i-1，j），cur_left），可以从当前行确定 cur_left

right（i，j）= min（right（i-1，j），cur_right），可以从当前行确定 cur_right

如果矩阵[i] [j] =='1'，height（i，j）= height（i-1，j）+1。

如果矩阵[i] [j] =='0'，那么 height（i，j）= 0，

如果你认为这个算法不容易理解，你可以试试这个例子：

0 0 0 1 0 0 0

0 0 1 1 1 0 0

0 1 1 1 1 1 0

从 0 行到 2 行的向量“左”和“右”如下

第 0 行：	第 1 行：	第 2 行：
1：0 0 0 3 0 0 0	1：0 0 2 3 2 0 0	l：0 1 2 3 2 1 0
r：7 7 7 4 7 7 7	r：7 7 5 4 5 7 7	r：7 6 5 4 5 6 7

向量“左”正在计算左边界。 以（i，j）=（1,3）为例。 在当前行 1 上，左边界在 j = 2 处。 但是，由于矩阵[1] [3]为 1，所以还需要考虑前一行的左边界，即 3。因此，（1,3）处的实际左边界为 3。

代码如下。 循环可以结合速度，但为了更清晰的算法，我将它们分开。

```
#include<vector>
#include<algorithm>
#include<iostream>
using namespace std;
int maximalRectangle(vector<vector<char> >&matrix){
    if(matrix.empty())  return 0;
    const int m=matrix.size();
    const int n=matrix[0].size();
    int left[n],right[n],height[n];
    fill_n(left,n,0);fill_n(right,n,n);fill_n(height,n,0);
    int maxA=0;
    for(int i=0;i<m;i++){
        int cur_left=0,cur_right=n;
```

```
        for(int j=0;j<n;j++){
            if(matrix[i][j]=='1')   height[j]++;
            else    height[j]=0;
        }
        for(int j=0;j<n;j++){
            if(matrix[i][j]=='1')   left[j]=max(left[j],cur_left);
            else{
                left[j]=0;
                cur_left=j+1;
            }
        }
        for(int j=n-1;j>=0;j--){
            if(matrix[i][j]=='1')   right[j]=min(right[j],cur_right);
            else{
                right[j]=n;
                cur_right=j;
            }
        }
        for(int j=0;j<n;j++)
            maxA=max(maxA,(right[j]-left[j])*height[j]);
    }
    return maxA;
}
int main(void){
    int M=4,N=5;
    char b[M][N]={{'1','0','1','0','0'},
                  {'1','0','1','1','1'},
                  {'1','1','1','1','1'},
                  {'1','0','0','1','0'}};
    vector<vector<char> > MATRIX(M,vector<char>(N));
    for(int i=0;i<M;i++)
        for(int j=0;j<N;j++)
            MATRIX[i][j]=b[i][j];
    std::cout<<maximalRectangle(MATRIX);    //6
}
```

25.字符串的编辑距离.Cpp

给定两个单词 word1 和 word2，找到将 word1 转换为 word2 所需的最小步数。 （每个操作都算作 1 步。）

允许你对一个单词做以下 3 个操作：

a）插入一个字符　　b）删除一个字符　　c）替换一个字符

使用 f [i] [j]表示 word1 [0，i）和 word2 [0，j）之间的最短编辑距离。 然后比较 word1 [0，i）和 word2 [0，j）的最后一个字符，分别是 c 和 d（c == word1 [i-1]，d == word2 [j-1]）：

如果 c == d，那么：f [i] [j] = f [i-1] [j-1]

否则，我们可以使用三个操作将 word1 转换为 word2：

（a）如果用 d 代替 c：　f [i] [j] = f [i-1] [j-1] + 1;

（b）在 c 之后加上 d：　f [i] [j] = f [i] [j-1] + 1;

（c）如果我们删除了 c：f [i] [j] = f [i-1] [j] + 1;

请注意，f [i] [j]仅取决于 f [i-1] [j-1]，f [i-1] [j]和 f [i] [j-1]，因此可以减少空间复杂度到 O（n） 通过仅使用第（i-1）个阵列和先前更新的元素（f [i] [j-1]）。

```
#include<string>
#include<algorithm>
```

```
#include<iostream>
#include<vector>
using namespace std;
int minDistance(string word1,string word2){
    int l1=word1.size();
    int l2=word2.size();
    vector<int> f(l2+1,0);
    for(int j=1;j<=l2;j++)
        f[j]=j;
    for(int i=1;i<=l1;i++){
        int prev=i;
        for(int j=1;j<=l2;j++){
            int cur;
            if(word1[i-1]==word2[j-1])
                cur=f[j-1];
            else
                cur=min(min(f[j-1],prev),f[j])+1;
            f[j-1]=prev;
            prev=cur;
        }
        f[l2]=prev;
    }
    return f[l2];
}
int main(void){
    string a="abcdefg";
    string b="acegkr";
    cout<<minDistance(a,b);     //5
}
```

矩阵的旋转操作：

```
/*
 * clockwise rotate
 * first reverse up to down, then swap the symmetry
 * 1 2 3     7 8 9     7 4 1
 * 4 5 6  => 4 5 6  => 8 5 2
 * 7 8 9     1 2 3     9 6 3
*/
void rotate(vector<vector<int> > &matrix) {
    reverse(matrix.begin(), matrix.end());
    for (int i = 0; i < matrix.size(); ++i) {
        for (int j = i + 1; j < matrix[i].size(); ++j)
            swap(matrix[i][j], matrix[j][i]);
    }
}
/*
 * anticlockwise rotate
 * first reverse left to right, then swap the symmetry
 * 1 2 3     3 2 1     3 6 9
 * 4 5 6  => 6 5 4  => 2 5 8
 * 7 8 9     9 8 7     1 4 7
*/
void anti_rotate(vector<vector<int> > &matrix) {
    for (auto vi : matrix) reverse(vi.begin(), vi.end());
    for (int i = 0; i < matrix.size(); ++i) {
```

```
        for (int j = i + 1; j < matrix[i].size(); ++j)
            swap(matrix[i][j], matrix[j][i]);
    }
}
```

DFS 实现全排列：

```
vector<vector<int> > permute(vector<int> &num) {
    vector<vector<int> > result;
    permuteRecursive(num, 0, result);
    return result;
}
                                // permute num[begin..end]
                    // invariant: num[0..begin-1] have been fixed/permuted
void permuteRecursive(vector<int> &num, int begin, vector<vector<int> >
&result){
    if (begin >= num.size()) {
                                        // one permutation instance
        result.push_back(num);
        return;
    }
    for (int i = begin; i < num.size(); i++) {
        swap(num[begin], num[i]);
        permuteRecursive(num, begin + 1, result);
                                                    // reset
        swap(num[begin], num[i]);
    }
}
```

不使用加减符号计算两个整数的和：

```
    int getSum(int a, int b) {
        return b==0?a:getSum(a^b,(a&b)<<1);
    }
```

11.k 个最频繁的元素.java

给定一个非空数组，返回 k 个最频繁的元素。例如，给定[1,1,1,2,2,3]和 k = 2，返回[1,2]。注意： 你可以假设 k 总是有效的，1≤k≤唯一元素的个数。你的算法的时间复杂度必须比 O（n log n）好，其中 n 是数组的大小。

建立一个列表数组，作为长度为 1 的桶进行排序。

```
import java.util.*;
public class Helloworld {
    public static List<Integer> topKFrequent(int []nums, int k){
        List<Integer>[] bucket = new List[nums.length+1];
        Map<Integer,Integer> frequencyMap = new HashMap<Integer,Integer>();
        for(int n : nums)
            frequencyMap.put(n, frequencyMap.getOrDefault(n, 0)+1);
        for(int key : frequencyMap.keySet()){
            int frequency = frequencyMap.get(key);
            if(bucket[frequency]==null)
                bucket[frequency] = new ArrayList<>();
            bucket[frequency].add(key);
        }
        List<Integer> res = new ArrayList<>();
        for(int pos=bucket.length-1;pos>=0 && res.size()<k;pos--)
            if(bucket[pos]!=null)
                res.addAll(bucket[pos]);
        return res;
    }
    public static void main(String[] args){
        int[] a={1,1,1,2,2,3};
        System.out.println(topKFrequent(a,2));
    }
```

}

生成集合的所有子集：

每个数字只有两种可能性：在或者不在一个子集里面。 这可以用 0 或者 1 来表示。

还有另一种方法来形象化这个想法。 也就是说，如果我们使用集合[1,2,3]，每两个连续的子集中出现一次 1，每四个连续的子集出现两次，每 8 个子集出现四次，如下所示（最初 8 个子集都是空的）：

[], [], [], [], [], [], [], []

[], [1], [], [1], [], [1], [], [1]

[], [1], [2], [1, 2], [], [1], [2], [1, 2]

[], [1], [2], [1, 2], [3], [1, 3], [2, 3], [1, 2, 3]

```
def subsets(nums):
    res = []
    nums.sort()
    for i in xrange(1<<len(nums)):
        tmp = []
        for j in xrange(len(nums)):
            if i & 1 << j:            # if i >> j & 1:
                tmp.append(nums[j])
        res.append(tmp)
    return res
```

最大的数字：

给定一个非负整数列表，排列它们，使它们成为最大的数字。

例如，给定[3，30，34，5，9]，最大的形成数字是 9534330。

注意：结果可能非常大，所以你需要返回一个字符串而不是一个整数。

逻辑非常简单。 只需比较数字通过将其转换为字符串。

```java
public String largestNumber(int[] num) {
    String[] array = Arrays.stream(num).mapToObj(String::valueOf).toArray(String[]::new);
    Arrays.sort(array, (String s1, String s2) -> (s2 + s1).compareTo(s1 + s2));
    return Arrays.stream(array).reduce((x, y) -> x.equals("0") ? y : x + y).get();
}
```

↑

Java 版

```python
def largestNumber(self, num):
    num = [str(x) for x in num]
    num.sort(cmp=lambda x, y: cmp(y+x, x+y))
    return ''.join(num).lstrip('0') or '0'
```

←Python 版

C++版

↓

```cpp
string largestNumber(vector<int> &num) {
    vector<string> arr;
    for(auto i:num)
        arr.push_back(to_string(i));
    sort(begin(arr), end(arr), [](string &s1, string &s2){ return s1+s2>s2+s1; });
    string res;
    for(auto s:arr)
        res+=s;
    while(res[0]=='0' && res.length()>1)
        res.erase(0,1);
    return  res;
}
```

12.除了自己之外的数组累乘.java

给定一个 n 个整数的数组 nums，其中 n> 1，返回一个数组输出，使得输出[i]等于除 nums [i]之外的所有 num 的元素的乘积。要求不能使用除法且时间复杂度为 O（n）。
例如，给出[1,2,3,4]，返回[24,12,8,6]。
这个想法很简单。累乘是使用当前数字之前的数字和当前数字之后的数字来计算的。因此，我们可以扫描数组两次。 首先，我们计算当前数字之前的数字的运行累乘。 其次，我们通过从数组尾部扫描来计算当前数字之后部分的运行累乘。

```
public int[] productExceptSelf(int[] nums) {
    int n = nums.length;
    int[] res = new int[n];
    res[0] = 1;
    for (int i = 1; i < n; i++) {
        res[i] = res[i - 1] * nums[i - 1];
    }
    int right = 1;
    for (int i = n - 1; i >= 0; i--) {
        res[i] *= right;
        right *= nums[i];
    }
    return res;
}
```

更多面试题请登录 https://leetcode.com/problemset/top-interview-questions/

36.一些科学计算的例子.py

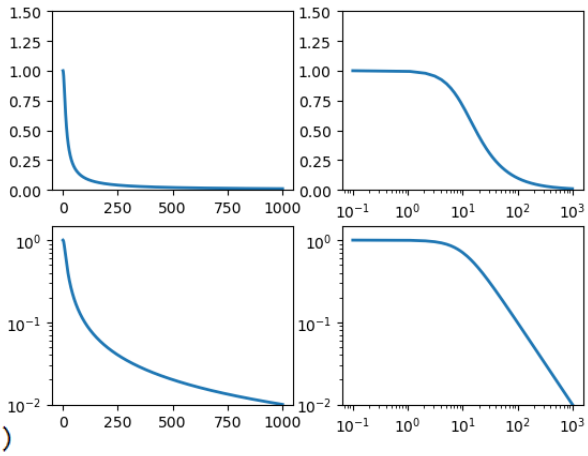
```
#coding=utf-8
from scipy import constants as C
print(C.h)
print(C.physical_constants['electron mass'])            #物理学常数
import scipy.special as S
print(S.gamma(1+1j))                                      #特殊函数：gamma 函数
from math import sin
from scipy import optimize as om
def f(x):
    x0, x1, x2 = x.tolist()
    return [5*x1+3, 4*x0*x0-2*sin(x1*x2), x1*x2-1.5]
result = om.fsolve(f, [1, 1, 1])                          #求解非线性方程组
print(result)
import numpy as np
X = np.array([8.19, 2.72, 6.39, 8.71, 4.7, 2.66, 3.78])
Y = np.array([7.01, 2.78, 6.47, 6.71, 4.1, 4.23, 4.05])
def residuals(p):
    k, b = p
    return Y - (k*X+b)
r = om.leastsq(residuals, [1, 0])
k, b = r[0]
print('k=', k, 'b=', b)                                 #最小二乘拟合
myMatrix = np.random.randint(0, 10, (3, 3))
print(myMatrix)
import scipy.linalg as ln
print(ln.svd(myMatrix))                                       #矩阵的奇异值分解
from scipy import sparse as pa
E = pa.lil_matrix((5, 5))
E[2, 3], E[3, 4], E[3, 2] = 5.5, 6.6, 7.7                     #稀疏矩阵
print(E.toarray())
w = pa.dok_matrix((4, 4))                             #4 个结点的带权有向图
edges = [(0, 1, 10), (1, 2, 5), (0, 2, 3), (2, 3, 7), (3, 0, 4), (3, 2, 6)]
for i, j, v in edges:
    w[i, j] = v
print(w.todense())                                          #图的邻接矩阵
```

输出结果：

```
6.62607004e-34
(9.10938356e-31, ‘kg’, 1.1e-38)
(0.498015668118-0.154949828302j)
[-0.70622057 -0.6        -2.5       ]
(‘k=’, 0.61349534919304416, ‘b=’, 1.7940925432593871)
[[9 7 5]
 [7 6 5]
 [3 1 9]]
(array([[-0.68697752,  0.38881725,  0.61390799],
       [-0.58580328,  0.20358353, -0.78446686],
       [-0.42999581, -0.89854042,  0.08791311]]), array([ 17.75675439,   6.36411088,
0.44245456]), array([[-0.65177497, -0.49297625, -0.57633654],
       [ 0.35021683,  0.47841428, -0.80527507],
       [ 0.67270912, -0.72670089, -0.13916988]]))
[[ 0.   0.   0.   0.   0. ]
 [ 0.   0.   0.   0.   0. ]
 [ 0.   0.   0.   5.5  0. ]
 [ 0.   0.   7.7  0.   6.6]
 [ 0.   0.   0.   0.   0. ]]
[[  0.  10.   3.   0.]
 [  0.   0.   5.   0.]
 [  0.   0.   0.   7.]
 [  4.   0.   6.   0.]]
```

37. 低通滤波器的频率响应曲线(对数坐标图).py

```
from pylab import*
w = np.linspace(0.1, 1000, 1000)
p = np.abs(1/(1+0.1j*w))
fig, axes = plt.subplots(2, 2)
functions = ("plot", "semilogx", "semilogy", "loglog")
for ax, fname in zip(axes.ravel(), functions):
    func = getattr(ax, fname)
    func(w, p, linewidth=2)
    ax.set_ylim(0, 1.5)
show()                          #
```



现在继续前面的 sympy 符号运算

最简形式的分数：

```
In [4]: Rational(3,1239)
Out[4]:
```

$$\frac{1}{413}$$

分母有理化：

```
In [6]: radsimp(1/(y*sqrt(x)+x*sqrt(y)))
Out[6]:
```

$$\frac{-\sqrt{x}y+x\sqrt{y}}{xy(x-y)}$$

因式分解：

```
In [7]: factor(15*x**2+2*y-3*x-10*x*y)
Out[7]:
```

$$(3x-2y)(5x-1)$$

高阶偏导数：

```
In [8]: diff(sin(x*y),x,2,y,3)
Out[8]:
```

$$x(x^2y^2\cos(xy)+6xy\sin(xy)-6\cos(xy))$$

三角函数展开：

```
In [9]: expand(sin(2*x+y),trig=True)
Out[9]:
```

$$2\sin(x)\cos(x)\cos(y)+2\sin(y)\cos^2(x)-\sin(y)$$

泰勒展开：

```
In [10]: series(sqrt(x),x,4,5)
Out[10]:
```

$$1-(x-4)^2/64+(x-4)^3/512-5(x-4)^4/16384+x/4+\mathcal{O}((x-4)^5;x\to 4)$$

求解一阶微分方程：

```
In [11]: eq = Eq(f(x).diff(x), -(y*(x*y+1))/(x*(1+x*y+(x*y)**2)))

In [12]: dsolve(eq,f(x),hint="lie_group")
Out[12]:
```

$$f(x) = C_1 - y\log(x) + y\log\left(2x + 1/y - \sqrt{3}i/y\right)/2 + \sqrt{3}iy\log\left(2x + 1/y - \sqrt{3}i/y\right)/6 + y\log\left(2x + 1/y + \sqrt{3}i/y\right)/2 - \sqrt{3}iy\log\left(2x + 1/y + \sqrt{3}i/y\right)/6 - y\log(2)$$

分析时间性能：

```
In [14]: %timeit radsimp(1/(y*sqrt(x)+x*sqrt(y)))
100 loops, best of 3: 18.3 ms per loop
```

```
from collections import Counter
a = list('feadcecaefcbdcabdaefdeacdfcdbfefe')     # top 3 frequent elements
print(list((zip(*Counter(a).most_common(3)))[0]))   # ['e', 'c', 'd']
from fractions import Fraction
x = 3.141592653
y = Fraction(*x.as_integer_ratio())     # convert float to fraction
print(y)                          # 884279718837543/281474976710656
```

13.LRU cache.java

设计并实现最近最少使用（LRU）缓存的数据结构。 它应该支持以下操作：获取和放置。

get(key) - 如果键存在于缓存中，则获取键的值（默认总是正数），否则返回-1。

put(key,value) - 如果键不存在，则设置或插入值。 当缓存达到其容量时，在插入新项目之前应该使最近最少使用的项目无效。

例如：

```
LRUCache cache = new LRUCache( 2 /* capacity */ );
cache.put(1, 1);
cache.put(2, 2);
cache.get(1);       // returns 1
cache.put(3, 3);    // evicts key 2
cache.get(2);       // returns -1 (not found)
cache.put(4, 4);    // evicts key 1
cache.get(1);       // returns -1 (not found)
cache.get(3);       // returns 3
cache.get(4);       // returns 4
```

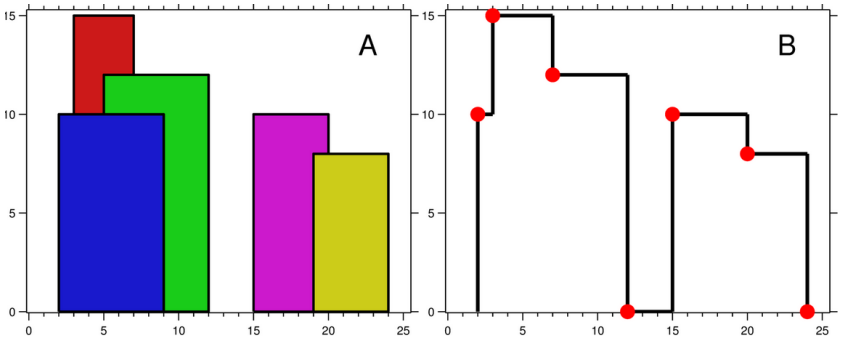
可以使用双向链表 + HashMap 来实现：

```
class LRUCache {
    class DLinkedNode{
        int key;
        int value;
        DLinkedNode pre;
        DLinkedNode next;
    }
private void addNode(DLinkedNode node){
                                //Always add the new node right after the head
        node.pre=head;
        node.next=head.next;
        head.next.pre=node;
        head.next=node;
    }
    private void removeNode(DLinkedNode node){
        DLinkedNode pre=node.pre;
        DLinkedNode next=node.next;
        pre.next=next;
        next.pre=pre;
    }
```

```
    private void moveToHead(DLinkedNode node){
        this.removeNode(node);
        this.addNode(node);
    }
    private DLinkedNode popTail(){
        DLinkedNode res=tail.pre;
        this.removeNode(res);
        return res;
    }
    private HashMap<Integer,DLinkedNode> cache=new HashMap<>();
    private int count,capacity;
    private DLinkedNode head,tail;
    public LRUCache(int capacity) {
        this.count=0;
        this.capacity=capacity;
        head=new DLinkedNode();
        head.pre=null;
        tail=new DLinkedNode();
        tail.next=null;
        head.next=tail;
        tail.pre=head;
    }
    public int get(int key) {
        DLinkedNode node=cache.get(key);
        if(node==null)
            return -1;
        this.moveToHead(node);
        return node.value;
    }
    public void put(int key, int value) {
        DLinkedNode node=cache.get(key);
        if(node==null){
            DLinkedNode newNode=new DLinkedNode();
            newNode.key=key;
            newNode.value=value;
            this.cache.put(key,newNode);
            this.addNode(newNode);
            ++count;
            if(count>capacity){
                DLinkedNode tail=this.popTail();
                this.cache.remove(tail.key);
                --count;
            }
        }else{
            node.value=value;
            this.moveToHead(node);
        }
    }
}
```

14.城市的天际线.java

输入样例：[[2 9 10], [3 7 15], [5 12 12], [15 20 10], [19 24 8]]输出样例：[[2 10], [3 15], [7 12], [12 0], [15 10], [20 8], [24, 0]]



```
    public List<int[]> getSkyline(int[][] buildings) {
        List<int[]> res=new ArrayList<>();
        List<int[]> heights=new ArrayList<>();
                        //use a int[][] to collect all [start point,-height]
                                //and [end point,height] for every building
        for(int[] b:buildings){
            heights.add(new int[]{b[0],-b[2]});
                            //we know whether current point is a start point
            heights.add(new int[]{b[1],b[2]});
                            //or a end point based on the sign of its height
        }
        Collections.sort(heights,(h1,h2)->{
            if(h1[0]!=h2[0])      //sort it, firstly based on the first value
                return h1[0]-h2[0];
            return h1[1]-h2[1];         //then use the second to break ties
        });
        Queue<Integer> pq=new PriorityQueue<>((x1,x2)->(x2-x1));
                                    //Use a maxHeap to store possible heights
        pq.offer(0);      //Provide a initial value to make it more consistent
        int prev=0;           //Before starting, the previous max height is 0
        for(int[] height:heights){
            if(height[1]<0)
                pq.offer(-height[1]);         // a start point, add height
            else
                pq.remove(height[1]);         // a end point, remove height
            int cur=pq.peek();
            if(cur!=prev){
        //compare current max height with previous max height, update result and
                res.add(new int[]{height[0],cur});
                                            // previous max height if necessary
                prev=cur;
            }
        }
        return res;
    }
```

Java 实现的针对小写字母的字典树(Trie,前缀树):

```
class TrieNode{
    public char val;
    public boolean isword;
    public TrieNode[] children=new TrieNode[26];
    public TrieNode(){}
    TrieNode(char c){
        TrieNode node=new TrieNode();
        node.val=c;
    }
}
public class Trie {
    private TrieNode root;       /** Initialize your data structure here. */
    public Trie() {
        root=new TrieNode();
        root.val=' ';
    }                                /** Inserts a word into the trie. */
    public void insert(String word) {
        TrieNode p=root;
        for(int i=0;i<word.length();i++){
```

```
            char ch=word.charAt(i);
            if(p.children[ch-'a']==null)
                p.children[ch-'a']=new TrieNode(ch);
            p=p.children[ch-'a'];
        }
        p.isword=true;
    }                               /** Returns if the word is in the trie. */
    public boolean search(String word) {
        TrieNode p=root;
        for(int i=0;i<word.length();i++){
            char ch=word.charAt(i);
            if(p.children[ch-'a']==null)
                return false;
            p=p.children[ch-'a'];
        }
        return p.isword;
    }   // Returns if there is any word in the trie that starts with the given prefix.
    public boolean startsWith(String prefix) {
        TrieNode p=root;
        for(int i=0;i<prefix.length();i++){
            char ch=prefix.charAt(i);
            if(p.children[ch-'a']==null)
                return false;
            p=p.children[ch-'a'];
        }
        return true;
    }
}/ * Your Trie object will be instantiated and called as such:
 * Trie obj = new Trie();
 * obj.insert(word);
 * boolean param_2 = obj.search(word);
 * boolean param_3 = obj.startsWith(prefix); */
```

深入理解 TCP 三次握手和四次挥手过程：

三次握手：

A:“喂，你听得到吗？”A->SYN_SEND

B:“我听得到呀，你听得到我吗？”应答与请求同时发出 B->SYN_RCVD | A->ESTABLISHED

A:“我能听到你，今天 balabala……”B->ESTABLISHED

四次挥手：

A:“喂，我不说了。”A->FIN_WAIT1

B:“我知道了。等下，上一句还没说完。Balabala…..”B->CLOSE_WAIT | A->FIN_WAIT2

B:”好了，说完了，我也不说了。”B->LAST_ACK

A:”我知道了。”A->TIME_WAIT | B->CLOSED

A 等待 2MSL,保证 B 收到了消息,否则重说一次”我知道了”,A->CLOSED

班上有 N 名学生。有些是朋友，有些则不是。他们的友谊是传递性的。例如，如果 A 是 B 的直接朋友，而 B 是 C 的直接朋友，那么 A 是 C 的间接朋友。我们定义的朋友圈是一群直接或间接的朋友。给定 N * N 矩阵 M 代表班级中学生之间的朋友关系。如果 M [i] [j] = 1，那么第 i 和第 j 个学生是彼此的直接朋友，否则不是。输出所有学生中的朋友圈的总数

Python 实现的两个版本：

```
import scipy.sparse
def findCircleNum(self, M):
    return scipy.sparse.csgraph.connected_components(M)[0]
import numpy as np
def findCircleNum(self, M):
    return len(set(map(tuple, (np.matrix(M, dtype='bool')**len(M)).A)))
```

后记

这本笔记里面包含了一个 CS 专业的本科毕业生应该清楚的全部基础知识，这本书应该经常翻看并且做到完全吸收滚瓜烂熟，否则记笔记也就失去了意义。

笔记中欠缺的知识，去工作实践中学习，还需要认真读的书有（包括但不限于）：

《Github 入门与实践》
《鸟哥的 Linux 私房菜》
《深入理解计算机系统》
《算法导论》

传统开发岗位：
《C++ primer》
《Java 编程思想》

大数据/机器学习/算法方向：
《Python 网络数据采集》
《Spark 快速大数据分析》
《集体智慧编程》
《SQL 必知必会》
《高性能 MySQL》
《统计学习方法》
《Python 机器学习及实践》
《R 语言实战》

保持终身自学的习惯，才能保证自己不被淘汰。不断学习新东西！
学会用 Google

最后：
Talk is cheap,show me the code