第一章习题

2．在欧几里德提出的欧几里德算法中（即最初的欧几里德算法）用的不是除法而是减法。请用伪代码描述这个版本的欧几里德算法

解：

1.r=m-n
2.循环直到 r=0
2.1 m=n
2.2 n=r
2.3 r=m-n
3 输出 m

4． 设数组 a[n]中的元素均不相等，设计算法找出 a[n]中一个既不是最大也不是最小的元素，并说明最坏情况下的比较次数。要求分别给出伪代码和 C++描述。

解：

```
#include<iostream>
using namespace std;

int main(){
        int a[]={1,2,3,6,4,9,0};
        int mid_value=0;//将“既不是最大也不是最小的元素”的值赋值给它
        for(int i=0;i<=4;++i){
            if(a[i+1]>a[i]&&a[i+1]<a[i+2]){
                        mid_value=a[i+1];
                        cout<<mid_value<<endl;
                        break;
            }
            else if(a[i+1]<a[i]&&a[i+1]>a[i+2]){
                        mid_value=a[i+1];
                        cout<<mid_value<<endl;
                        break;
            }
       }
        return 0;
}
```

最坏情况下的比较次数与数组分布有关，与算法的设计也有关

例：10 11 7 8 6　数组按此类型分布时，一共需要比较 2*（n-2）次

第二章习题

1．如果 $T_1(n)=O(f(n))$，$T_2(n)=O(g(n))$，解答下列问题：

（1）证明加法定理：$T_1(n)+T_2(n)=\max\{O(f(n)), O(g(n))\}$；

（2）证明乘法定理：$T_1(n)\times T_2(n)=O(f(n))\times O(g(n))$；

（3）举例说明在什么情况下应用加法定理和乘法定理。

解：

（1）（2）证明思路：根据教材 P19 定义 2.1 对（1）和（2）证明

（3）比如在

for（f(n)）{

for(g(n))

} 中应该用乘法定理

如果在“讲两个数组合并成一个数组时”，应当用加法定理

2．考虑下面的算法，回答下列问题：算法完成什么功能？算法的基本语句是什么？基本语句执行了多少次？算法的时间复杂性是多少？

（1）
```
int Stery(int n)
{
  int S = 0;
  for (int i = 1; i <= n; i++)
    S = S + i * i;
  return S;
}
```

（2）
```
int Q(int n)
{
  if (n == 1)
    return 1;
  else
    return Q(n−1) + 2 * n − 1;
}
```

解：

（1） 完成的是 1~n 的平方和

基本语句：S=S+i*i，执行了 n 次

时间复杂度 O（n）

（2） 完成的是 n 的平方

基本语句：return Q(n-1) + 2 * n – 1，执行了 n 次

时间复杂度 O（n）

3. 分析以下程序段中基本语句的执行次数是多少，要求列出计算公式。

（1）
```
for (i = 1; i <= n; i++)
  if (2*i <= n)
    for (j = 2*i; j <= n; j++)
      y = y + i * j;
```

（2）
```
m = 0;
for (i = 1; i <= n; i++)
  for (j = 1; j <= 2*i; j++)
    m=m+1;
```

解：

(1) 基本语句 y = y + i * j 执行了 $\sum_{i=1}^{\lfloor n/2 \rfloor}(n-2i+1)$ 次，时间复杂度为 $O(n^2)$

(2) 基本语句 m+=1 执行了 $\sum_{i=1}^{n} 2i$ 次，时间复杂度为 $O(n^2)$

4. 使用扩展递归技术求解下列递推关系式：

（1） $T(n)=\begin{cases} 4 & n=1 \\ 3T(n-1) & n>1 \end{cases}$　　（2） $T(n)=\begin{cases} 1 & n=1 \\ 2T(n/3)+n & n>1 \end{cases}$

解：（1）T（n）=3T(n-1)=3^2*T(n-2)=.....=3^{n-1}T(1)=4*3^{n-1} =O(3^n)

(2) 利用定理 2.1，则有：a=2, b=3, k=1, $a<b^k$, 故 T(n)=O(n)

5. 求下列问题的平凡下界，并指出其下界是否紧密。

（1）求数组中的最大元素；

（2）判断邻接矩阵表示的无向图是不是完全图；

（3）确定数组中的元素是否都是唯一的；

（4）生成一个具有 *n* 个元素集合的所有子集

解：

(1) Ω(n) 紧密

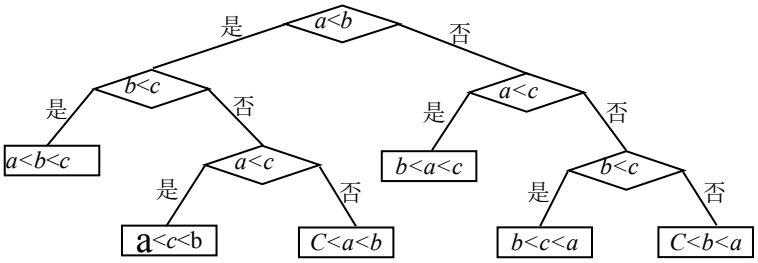
(2) Ω(*n*n)*

(3) Ω(nlog$_2$n+n)（先进行快排，然后进行比较查找）

(4) Ω(*2^n*)

7．画出在三个数 *a*, *b*, *c* 中求中值问题的判定树。

解：



第三章习题

1．假设在文本"ababcabccabccacbab"中查找模式"abccac"，写出分别采用 BF 算法和 KMP 算法的串匹配过程

```
//BF 算法
#include<iostream>
using namespace std;

int BF(char S[], char T[])
{
   int index = 0;
int i = 0, j = 0;
while ((S[i] != '\0') && (T[j] != '\0'))
{
        if (S[i] == T[j])
        {
            i++;
            j++;
        }
```

```
        else {
              ++index;
              i = index;
              j = 0;
        }
       }
   if (T[j] == '\0')
        return index + 1;
         else
              return 0;
}
int main()
{
      char s1[19]="ababcabccabccacbab";
      char s2[7]="abccac";
      cout<< BF( s1, s2) <<endl;
      return 0;

}

//KMP 算法
#include<iostream>
using namespace std;

void GetNext(char T[ ], int next[ ])              //求模式 T 的 next 值
{
int i, j, len;
next[0] = -1;
   for (j = 1; T[j]!='\0'; j++)                   //依次求 next[j]
{
for (len = j - 1; len >= 1; len--)             //相等子串的最大长度为 j-1
      {
        for (i = 0; i < len; i++)             //依次比较 T[0]~T[len-1]与 T[j-len]~T[j-1]
           if (T[i] != T[j-len+i]) break;
if (i == len)
{
           next[j] = len; break;
}
      }//for
      if (len < 1)
      next[j] = 0;                        //其他情况，无相等子串
   }//for
}
```

```
int KMP(char S[ ], char T[ ])                    //求 T 在 S 中的序号
{
int i = 0, j = 0;
int next[80];                                    //假定模式最长为 80 个字符
   GetNext(T, next);
   while (S[i] != '\0' && T[j] != '\0')
   {
if (S[i] == T[j])
{
i++; j++;
       }
       else {
          j = next[j];
          if (j == -1) {i++; j++;}
       }
}
   if (T[j] == '\0') return (i - strlen(T) +1);          //返回本趟匹配的开始位置
     else
     return 0;

}


int main()
{
       char s1[]="ababcabccabccacbab";
       char s2[]="abccac";
cout<<KMP(s1[],s2[])<<endl;

   return 0;

}
```

2．分式化简。设计算法，将一个给定的真分数化简为最简分数形式。例如，将 6/8 化简为 3/4。

```
#include<iostream>
using namespace std;

int main()
{
   int n;//分子
   int m;//分母
```

```
    int factor;//最大公因子
    int factor1;
    cout<<"输入一个真分数的分子与分母：  "<<endl;
    cin>>n>>m;

    int r = m % n;//因为是真分数  所以分母一定大于分子
        factor1=m;
        factor=n;
while (r != 0)
{

        factor1 =factor;
        factor = r;
        r = factor1% factor;
}
cout<<"输出该真分数的最简分数：  "<<(n/factor)<<"/"<<(m/factor)<<endl;
     return 0;

}
```

3．设计算法，判断一个大整数能否被 11 整除。可以通过以下方法：将该数的十进制表示从右端开始，每两位一组构成一个整数，然后将这些数相加，判断其和能否被 11 整除。例如，将 562843748 分割成 5,62,84,37,48，然后判断(5+62+84+37+48)能否被 11 整除

```
//将一个大整数看成一个数组
//数组的奇数位对应数的 10 倍加上数组偶数对应数的本身
//验证结果能否被 11 整除

#include<iostream>
using namespace std;

int main()
{
        int a[9]={5,6,2,8,4,3,7,4,8};
        int result=0;    //result 为题目要求的各位之和
        for(int i=0;i!=9;++i)
        {
            if(i%2==0)
                  result+=a[i];    //i  为偶数位时，结果加上其对应数组数的本身
            else
                  result+=a[i]*10;    //i  为奇数位时，结果加上对应数组数的 10 倍
        }

        if(result%11==0)
              cout<<"该整数能被 11 整除"<<endl;
```

```
    else
        cout<<"该整数不能被 11 整除"<<endl;

    return 0;
}
```

6. 设计算法，在数组 *r*[*n*]中删除所有元素值为 *x* 的元素，要求时间复杂性为 *O*(*n*)，空间复杂性为 *O*(1)。

没有比此题更简单的题了，略。

7. 设计算法，在数组 *r*[*n*]中删除重复的元素，要求移动元素的次数较少并使剩余元素间的相对次序保持不变。

```
//在数组查找相同的元素
//把其中一个相同的数值的元素位置设成一个“特殊数值”
//输出所求函数

#include<iostream>
using namespace std;

int main()
{
    int a[]={1,2,1,5,3,2,9,4,5,5,3,5};
    int i,j;
    for( i=0;i<12;i++)
    {
        for(j=0;j<i;j++)
        {
            if(a[j]==a[i])
                a[i]=64787250;//设一个数组不存在的数值
        }
    }//for
    for(i=0;i<12;i++)
    {
        if(a[i]!=64787250)
            cout<<a[i]<<" ";
    }
            cout<<endl;

            return 0;
}
```

第四章习题

3、分治策略一定导致递归吗？如果是，请解释原因。如果不是，给出一个不包含递归的分治例子，并阐述这种分治和包含递归的分治的主要不同。

不一定导致递归。
如非递归的二叉树中序遍历。
这种分治方法与递归的二叉树中序遍历主要区别是：应用了栈这个数据结构。

4、对于待排序序列(5, 3, 1, 9)，分别画出归并排序和快速排序的递归运行轨迹。

1）归并排序：
第一趟：（5,3）（1,9）;
第二趟：（5）,（3）,（1）,（9）;
第三趟：（3,5）（1,9）;
第四趟：（1,3,5,9）;
2）快速排序：
第一趟：(5,3,1,9)；　　//5 为轴值，从右向左扫描，与 1 交换得(1,3,5,9)，然后对（3,5,9）从左向右扫描，碰到 5 停止，完成一次划分
第二趟：（1,3）, 5,（9）;　　//1 为轴值，对（1,3）从右向左扫描，完成一次划分
第三趟：1,（3）, 5,（9）;　　//3 为轴值，完成一次划分
第四趟：1,3, 5,（9）;　　//9 为轴值，完成一次划分
最终结果：（1,3,5,9）;

5、设计分治算法求一个数组中的最大元素，并分析时间性能。

```
//简单的分治问题
//将数组均衡的分为“前”，“后”两部分
//分别求出这两部分最大值，然后再比较这两个最大值

#include<iostream>
using namespace std;

extern const int n=6;//声明
int main()
{
    int a[n]={0,6,1,2,3,5};//初始化
    int mid=n;
    int num_max1=0,num_max2=0;
    for(int i=0;i<=n;++i)//前半部分
    {
      if(a[i]>num_max1)
          num_max1=a[i];
    }
    for(int j=n+1;j<n;++j)//后半部分
```

```
        {
            if(a[j]>num_max2)
              num_max2=a[j];
        }
        if(num_max1>=num_max2)
            cout<<"数组中的最大元素：  "<<num_max1<<endl;
        else
            cout<<"数组中的最大元素：  "<<num_max2<<endl;

        return 0;

    }
```

时间复杂度：O（n）

6、 设计分治算法，实现将数组 *A*[*n*]中所有元素循环左移 *k* 个位置，要求时间复杂性为 *O*(*n*)，空间复杂性为 *O*(1)。例如，对 *abcdefgh* 循环左移 3 位得到 *defghabc*。

```
 //采用分治法
//将数组分为 0-k-1 和 k-n-1 两块
//将这两块分别左移
//然后再合并左移

#include <iostream>
using namespace std;


    void LeftReverse(char *a, int begin, int end)
{
    for(int i=0;i<(end-begin+1)/2;i++)//交换移动
    {
    int temp=a[begin+i];
    a[begin+i]=a[end-i];
     a[end-i]=temp;
    }
}

    void Converse(char *a,int n,int k)
{
     LeftReverse(a, 0, k+1);
      LeftReverse(a, k, n+1);
     LeftReverse(a, 0, n-1);
      for(int i=0;i<n;i++)
          cout<<a[i]<<" ";
```

```
                cout<<endl;
}

      int main()
{
         char a[7]={'a','b','c','d','e','f','g'};
           Converse(a,7,3);

         return 0;

}
```

7、设计递归算法生成 *n* 个元素的所有排列对象。

```
#include <iostream>
using namespace std;

int data[100];

//在 m 个数中输出 n 个排列数（n<=m）
void DPpl(int num,int m,int n,int depth)
{
if(depth==n)
  {
for(int i=0;i<n;i++)
cout<<data[i]<<" ";
cout<<endl;
}
   for(int j=0;j<m;j++)
  {
   if((num&(1<<j))==0)
   {     data[depth]=j+1;
        DPpl(num+(1<<j),m,n,depth+1);
    }
  }//for
}
int main()
{
    DPpl(0,5,1,0);
    DPpl(0,5,2,0);
    DPpl(0,5,3,0);
    DPpl(0,5,4,0);
    DPpl(0,5,5,0);

      return 0;
```

}

第五章习题

1、下面这个折半查找算法正确吗？如果正确，请给出算法的正确性证明，如果不正确，请说明产生错误的原因。

```
int BinSearch(int r[ ], int n, int k)
{
   int low = 0, high = n - 1;
   int mid;
   while (low <= high)
   {
      mid = (low + high) / 2;
      if (k < r[mid]) high = mid;
      else  if (k > r[mid]) low = mid;
            else return mid;
   }
   return 0;
}
```

错误。

正确算法：

```
int BinSearch1(int r[ ], int n, int k)
{
   int low = 0, high = n - 1;
   int mid;
   while (low <= high)
   {
      mid = (low + high) / 2;
      if (k < r[mid]) high = mid - 1;
      else  if (k > r[mid]) low = mid + 1;
            else return mid;
   }
   return 0;
}
```

2、请写出折半查找的递归算法，并分析时间性能。

```
//折半查找的递归实现

#include<iostream>
using namespace std;

int digui_search(int a[],int low,int high,int x)
```

```
{
   if (low > high)
      return 0;

   int mid = (low+high)/2;
   if (a[mid] == x)
      return mid;
   else if (a[mid] < x)
      digui_search(a,low,mid-1,x);
   else
      digui_search(a,mid+1,high,x);
}
int main()
{
      int a[6]={0,1,2,9,5,3};
int result=digui_search(a,0,5,5);

  cout<<a[result]<<endl;
      return 0;

}
```

4. 求两个正整数 *m* 和 *n* 的最小公倍数。（提示：*m* 和 *n* 的最小公倍数 lcm(*m*, *n*)与 *m* 和 *n* 的最大公约数 gcd(*m*, *n*)之间有如下关系：lcm(*m*, *n*)=*m*×*n*/gcd(*m*, *n*)）

```
//求两个数的最小公倍数

#include<iostream>
using namespace std;

int main (void)
{
  int a,b;
    int i=1;

  cin>>a>>b;
  while((i%a!=0)||(i%b!=0))
        ++i;
  cout<<"a,b 最小公倍数为："<<i<<endl;

     return 0;
}
```

(该算法比较直接，要使其改进，可用欧几里得算法求得两个数的最大公约数，然后套用上面的公式再求最小公倍数)

5. 插入法调整堆。已知（$k_1, k_2, \ldots, k_n$）是堆，设计算法将（$k_1, k_2, \ldots, k_n, k_{n+1}$）调整为堆（假设调整为大根堆）。

参照：

```
    void SiftHeap(int r[ ], int k, int n)
{
  int i, j, temp;
  i = k; j = 2 * i + 1;                          //置 i 为要筛的结点，j 为 i 的左孩子
  while (j < n)                                  //筛选还没有进行到叶子
  {
    if (j < n-1 && r[j] < r[j+1]) j++;           //比较 i 的左右孩子，j 为较大者
    if (r[i] > r[j])                             //根结点已经大于左右孩子中的较大者
      break;
    else {
      temp = r[i]; r[i] = r[j]; r[j] = temp;     //将被筛结点与结点 j 交换
      i = j; j = 2 * i + 1;                      //被筛结点位于原来结点 j 的位置
    }
  }
}
```

进行调堆！