实用算法设计——查找

主讲：娄文启

louwenqi@ustc.edu.cn

5 查找

假定：待查找的数据已在内存中。

重点：理解如何将伪代码实现为正确的程序；掌握基本的代码调优技术。

难点：针对具体程序，把握：是否需要以及何时进行代码调优。

基础：C语言编程；线性结构的定义及基本操作的实现。

5 查找

5.1 基于索引表的查找

5.2 蛮力查找（顺序查找）

5.3 基于有序表的二分查找

5.4 字符串的查找

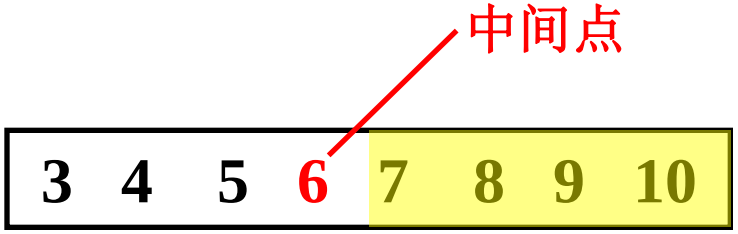
5.5 基于树的查找

5.3 基于有序表的二分查找

- 无处不在的二分查找（《编程珠玑》2.2节）
- 二分查找算法的基本设计思想
- 能正确实现二分查找算法很容易？（《编程珠玑》第5章）
  - 如何编写脚手架程序
  - 充分利用assert
  - 如何实现自动测试？
- 如何提高二分查找第一次出现位置的效率？（《编程珠玑》第9章）
- 二分查找很快？二分查找的应用场景？

无处不在的二分查找

- 问题：给定一个最多包含40亿个随机排列的32位整数的文件，找出一个不在文件中的32位整数。（在文件中必然缺失一个这样的数——为什么？）
  ① 在具有足够内存的情况下，如何解决该问题？
  ② 如果有几个外部的“临时”文件可用，但是仅有几百字节的内存，又该如何解决？（归并排序+二分搜索）



- 假设数组x中元素取值范围为[3,10]
- **Case 1:** x 3 4 5 10　　x中包含n=4个元素
  - *探测到中间点x[i]=6所在的位置*。
  - 大于中间点的元素有1个（<预期值4）
  - 小于等于中间点的元素有3个（<预期值4）
  - 因而数组x的左半部分区间和右半部分区间一定都缺失了元素。（大规模问题 => 小规模问题）
- **Case 2:** x 3 3 5 6 8 10　　x中包含n=6个元素
  - *探测到中间点x[i]=6所在的位置*
  - 大于中间点的元素有2个（<预期值4）
  - 小于等于中间点的元素有4个（=预期值4）
  - 因而数组x的右半部分区间一定缺失了元素。（大规模问题=>小规模问题）

注意：由于数组x中实际元素数目< 理论上的数目，故总可以找到一个区间缺失了元素，且该区间的范围可以逐渐缩减。

二分查找算法的基本设计思想

- **int BinarySearch(DataType t)：**
- 功能描述：在升序排列的线性表x中查找t出现的位置。
- 输入：要查找的目标项目t
- 输出：若查找失败，则输出-1；若查找成功，则输出匹配项的位置。
- 算法设计思路：

Step1: 设置查找区间为[0,n-1]

Step2: 若查找区间是合法的（下界<上界），则比较线性表x中查找区间内的中间元素x[(n-1)/2]与t，并进行相应处理

若中间元素x[(n-1)/2] < t，则将查找区间变为[(n-1)/2, (n-1)]，并重复Step2；

若中间元素x[(n-1)/2] > t，则将查找区间变为[0, (n-1)/2] ，并重复Step2 ；

若中间元素x[(n-1)/2] == t，则表示成功匹配，并输出匹配项的位置，退出程序。

Step3: 查找失败，输出-1。

正确实现二分查找算法

- **1946年，发表了第一篇二分查找论文；但是，16年后，第一个正确的二分查找程序才出现。**
- **错误实现的二分查找算法的代码示例：**

```
int binarysearch9(DataType t)
{   int l, u, m;
    l = 0;
    u = n-1;
    while (l <= u) {
        m = (l + u) / 2;
        if (x[m] < t)   //意味着t只可能在右半部分区间上
            l = m;
        else if (x[m] > t) //意味着t只可能在左半部分区间上
            u = m;
        else {   return m;  }  }
    return -1;  }
```

```
#define search binarysearch9

/* Scaffolding to probe one algorithm */
void probe1()
{    int i;
     DataType t;
     while (scanf("%d %d", &n, &t) != EOF) {
          for (i = 0; i < n; i++)
               x[i] = 10*i;
          printf(" %d\n", search(t));
     }
}

int main()
{    probe1();
     return 1;
}
```

编写脚手架程序来测试某个函数代码的正确性！

如何发现错误实现的二分查找算法的代码示例中的**bug**？

```
int binarysearch9(DataType t)
{    int l, u, m;
     l = 0;
     u = n-1;
     while (l <= u) {
          m = (l + u) / 2;
     printf("  %d %d %d\n", l, m, u);
          if (x[m] < t)
               l = m;
          else if (x[m] > t)
               u = m;
          else {
               return m;
          }
     }
     return -1;   //此时 l>u
}
```

利用断点调试技术

利用printf语句

如何发现错误实现的二分查找算法的代码示例中的bug？

```
int binarysearch9(DataType t)
{    int l, u, m;
     l = 0;
     u = n-1;
     while (l <= u) {
          m = (l + u) / 2;
          if (x[m] < t)
               l = m;
          else if (x[m] > t)
               u = m;
          else {
               assert(x[m] == t);
               return m;
          }
     }
     return -1;
}
```

```
#define assert(v) { if ((v) == 0) printf("
 binarysearch bug %d %d\n", i, n); }
```

利用assert语句，确保程序的输出是我们所预期的！

如何发现错误实现的二分查找算法的代码示例中的bug？

```
int binarysearch9(DataType t)
{    int l, u, m;
     int oldsize, size = n+1;
     l = 0;
     u = n-1;
     while (l <= u) {
          oldsize = size;
          size = u - l +1;
          assert(size < oldsize);
          m = (l + u) / 2;
          if (x[m] < t)
               l = m;
          else if (x[m] > t)
               u = m;
          else {     assert(x[m] == t);
               return m;          }
     }
     return -1;
}
```

```
#define assert(v) { if ((v) == 0) printf("
 binarysearch bug %d %d\n", i, n); }
```

利用assert语句，确保每次循环之后搜索范围都会变小！

找到了Bug！

对binarysearch2进行自动测试，验证其正确性。

Note：测试数据涵盖：

- 空数组：n=0;
- 成功查找:每个数组元素；
- 不成功的查找：各类不成功case。
  - 目标项不在数组中；
  - 目标项在数组中，但是不在搜索范围之内。

正确实现的二分查找算法的代码示例：

```
int binarysearch1(DataType t)
{    int l, u, m;
     l = 0;
     u = n-1;
     for (;;) {
          if (l > u)
            { //assert(( x[u]<t) && (x[u+1]>t));
              assert((u<0 || x[u]<t) && (u+1>=n || x[u+1]>t));
              return -1;}
          m = (l + u) / 2;
          if (x[m] < t)
                    l = m+1;
          else if (x[m] == t)
                    return m;
          else /* x[m] > t */
                    u = m-1;
     }}
```

正确实现的二分查找算法的代码示例：

```
int binarysearch2(DataType t)
{    int l, u, m;
     l = 0;
     u = n-1;
     while (l <= u) {
          m = (l + u) / 2;
          if (x[m] < t)
               l = m+1;
          else if (x[m] == t)
               return m;
          else /* x[m] > t */
               u = m-1;
     }
     return -1;
}
```

对binarysearch2进行自动测试，验证其正确性。

Note：测试数据涵盖：

- 空数组：n=0;
- 成功查找:每个数组元素;
- 不成功的查找：各类不成功case。
  - 目标项不在数组中;
  - 目标项在数组中，但是不在搜索范围之内。

```
#define s binarysearch2

/* Torture test one algorithm */
void test(int maxn)
{     int i;
      for (n = 0; n <= maxn; n++) {
            printf("n=%d\n", n);
            /* distinct elements (plus one at end) */
            for (i = 0; i <= n; i++)
                  x[i] = 10*i;
            for (i = 0; i < n; i++) {
                  assert(s(10*i)     ==  i);
                  assert(s(10*i - 5) == -1);          }
            assert(s(10*n - 5) == -1);
            assert(s(10*n)     == -1);
            /* equal elements */
            for (i = 0; i < n; i++)
                  x[i] = 10;
            if (n == 0) {  assert(s(10) == -1);   }
            else { assert(0 <= s(10) && s(10) < n);          }
            assert(s(5) == -1);
            assert(s(15) == -1);
      }
}
int main()
{     test(25);
      return 1;
}
```

利用assert语句自动测试!

避免测试数据的单一性：

- 空数组： n=0;
- 数组元素是否相异？
- 成功查找:每个数组元素；
- 不成功的查找：各类不成功case。
  - 目标项不在数组中；
  - 目标项在数组中，但是不在搜索范围之内。

```
#define MAXN 1000000
typedef int DataType;
DataType x[MAXN];
int sorted()
{   int i;
    for (i = 0; i < n-1; i++)
        if (x[i] > x[i+1])
            return 0;
    return 1;
}

int main()
{    int i;
     DataType t;
     while (scanf("%d %d", &n, &t) != EOF) {
            for (i = 0; i < n; i++)
                    x[i] = 10*i;
            assert(sorted());
            printf(" %d\n", binarysearch2(t));       }
     return 1;
}
```

利用assert语句判断二分查找是否被正确应用!

Note：assert语句判断是否有序时间开销较大，因此通常在数组初始化后，在所有二分查找之前只进行一次断言判断。

计时脚手架

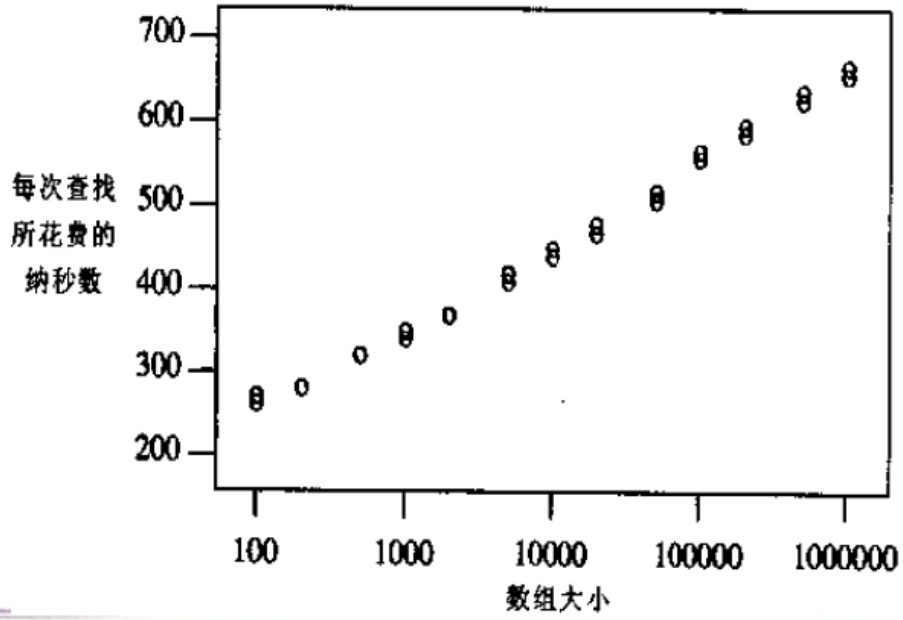
```
void timedriver()
{       int i, algnum, numtests, test, start, clicks;
        while (scanf("%d %d %d", &algnum, &n, &numtests) != EOF) {
                for (i = 0; i < n; i++)
                        x[i] = i;
                for (i = 0; i < n; i++)
                        p[i] = i;
                start = clock();
                for (test = 0; test < numtests; test++) {
                        for (i = 0; i < n; i++) {
                                switch (algnum) {
                                case 1:  assert(binarysearch1(p[i]) == p[i]); break;
                                case 2:  assert(binarysearch2(p[i]) == p[i]); break;
                                case 3:  assert(binarysearch3(p[i]) == p[i]); break;
                                case 4:  assert(binarysearch4(p[i]) == p[i]); break;
                                }       }       }
                clicks = clock() - start;
                printf("%d\t%d\t%d\t%d\t%g\n",
                        algnum, n, numtests, clicks,
                        1e9*clicks/((float) CLOCKS_PER_SEC*n*numtests));
        }
}

int main()
{       timedriver();
        return 1;
}
```

1、验证二分查找算法的时间开销是否为$O(\log_2 n)$？

2、是否所有情况下二分查找算法的时间开销都为$O(\log_2 n)$？

下面是与 400MHz 奔腾 II 中的程序进行的会话，与往常一样，斜体表示输入：

```
1 1000 10000
1       1000    10000   3445    344.5
1 10000 1000
1       10000   1000    4436    443.6
1 100000 100
1       100000  100     5658    565.8
1 1000000 10
1       1000000 10      6619    661.9
```



调试（边栏）

- 案例1：IBM的Yorktown Heights研究中心的一件“怪事”：
  - 一个程序员刚安装了一台新的工作站。当他坐着时一切正常；但是，当他站起来，就不能成功登录到系统中。（工作站如何知道程序员是坐着还是站着？）
- 提示：程序员坐着和站着时的登录有何区别？

- 案例2：芝加哥的一个银行系统已经正常运行了好几个月了，但是第一次用于国际数据就出现了非正常退出。
  - 进一步观察发现，当用户键入其首都的名字基多（Quito）时，程序将其解释为退出请求。
  - 经验：在程序退出前，你到底输入了什么？

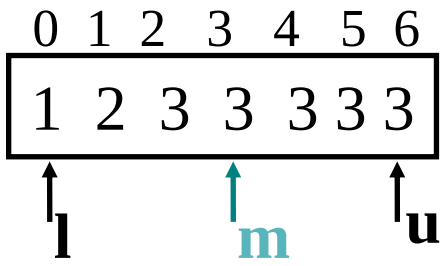
- 案例3：某系统出现“连续两轮仅首轮运行正确”。
  - 该系统能正确处理第一个事务，但是在随后的所有事务中，总是有小**bug**。当系统重新启动后，又能正确处理第一个事务，而在随后的所有事务又出现**bug**。
  - 问题所在：当程序加载时，变量的初始化是正确的；但是在第一个事务之后没有正确的复位。
  - 经验：程序在出错之前是否曾正确运行？正确运行了多少次？

- 调试是很困难的：调试所花费的时间可能是程序开发时间的**3-5**倍。
- 请记住：无论系统的行为咋看起来多么神秘莫测，其背后总有合乎逻辑的解释。
- 只要找到了正确的问题，那么就可以快速定位**bug**。

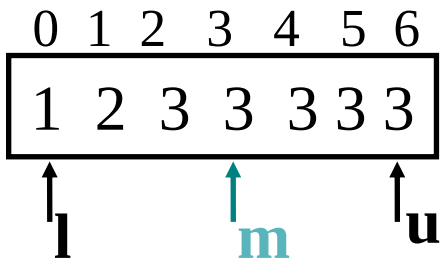
二分查找第一次出现的位置

- 二分查找通常不需要代码优化。
- 特殊需求：确定有序的整数序列x中，整数t第一次出现的位置。

```
int binarysearch1(DataType t)
{    int l, u, m;
     l = 0;
     u = n-1;
     for (;;) {
          if (l > u)
               return -1;
          m = (l + u) / 2;
          if (x[m] < t)
               l = m+1;
          else if (x[m] == t)
               return m;
          else /* x[m] > t */
               u = m-1;
     }
}
```



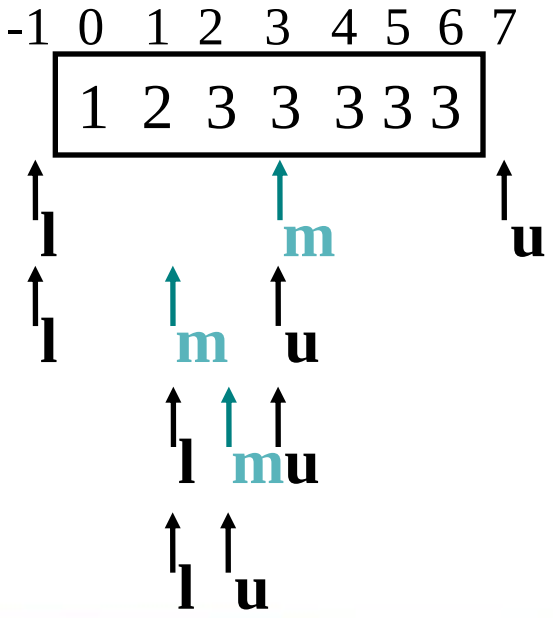
```
int binarysearch2(DataType t)
{    int l, u, m;
     l = 0;
     u = n-1;
     while (l <= u) {
          m = (l + u) / 2;
          if (x[m] < t)
               l = m+1;
          else if (x[m] == t)
               return m;
          else /* x[m] > t */
               u = m-1;
     }
     return -1;
}
```



二分查找第一次出现的位置

```
int binarysearch3(DataType t)
{   int l, u, m;
    l = -1;
    u = n;
    while (l+1 != u) {
        m = (l + u) / 2;
        if (x[m] < t)
            l = m;
        else
            u = m;
    }
    if (u >= n || x[u] != t)
        return -1;
    return u;
}
```

优化点：每次迭代循环中，t与x中的元素只作一次比较。



二分查找第一次出现的位置——优化

假设数组x包含n=1000个元素，使用[l,l+i]来表示查找范围，而不是使用[l,u]

```
i = 512;
l = -1;
if (x[511]   < t)   //若待查找项在数组的右半部分
        l = 1000 - 512;
while i !=1         //若待查找项在数组的左半部分
        nexti = i/2;
        if (x[l+nexti ] < t)
                l += nexti;
                i = nexti;
        else
                i = nexti;
p = l+1;
if (p >1000 || x[p] != t)
        p = -1;
return p;
```

二分查找第一次出现的位置——优化

假设数组x包含n=1000个元素，使用[l,l+i]来表示查找范围，而不是使用[l,u]

```
i = 512;
l = -1;
if (x[511]   < t)   //若待查找项在数组的右半部分
        l = 1000 - 512;
while i !=1         //若待查找项在数组的左半部分
        i = i/2;
        if (x[l+i] < t)
                l += i;



p = l+1;
if (p >1000 || x[p] != t)
        p = -1;
return p;
```

二分查找第一次出现的位置——优化

```
int binarysearch4(DataType t)
{    int l, p;
     if (n != 1000)
          return binarysearch3(t);
     l = -1;
     if (x[511]   < t) l = 1000 - 512;
     if (x[l+256] < t) l += 256;
     if (x[l+128] < t) l += 128;
     if (x[l+64 ] < t) l += 64;
     if (x[l+32 ] < t) l += 32;
     if (x[l+16 ] < t) l += 16;
     if (x[l+8  ] < t) l += 8;
     if (x[l+4  ] < t) l += 4;
     if (x[l+2  ] < t) l += 2;
     if (x[l+1  ] < t) l += 1;
     p = l+1;
     if (p >= n || x[p] != t)
          return -1;
     return p;
}
```

优化点：展开循环。

- n=1000时，最多进行10次循环

验证：程序真的正确吗？

```
assert(l<0 || x[l]<t);
if (x[l+256] < t) l += 256;
assert(l<0 || x[l]<t) ;
```

注意：该函数执行时，会执行10个if语句。这个与Case语句是不同的。

其他调优技术介绍

- 问题1：整数取模。
- 优化前：

```
k=(j+rotdist) % n;
```

- 优化后：

```
k =(j+rotdist);
while (k>n)
    k- = n;
```

通过将开销较大的模运算符用其他运算符来替换，可减少计算时间。

Note： 若I/O操作，或对内存的访问占用了大量时间，那么减少计算时间将是毫无意义的。

- 问题2：利用宏来替换函数。

```
float max(float a, float b)
{   return a>b? a:b;  }

#define max(a,b) ((a)>(b)? (a):(b));
```

- 但是，有可能会起到反作用。比如，递归调用的函数。

```
float maxsum3(l,u)
{   ……
    return max(maxsum3(l,m),maxsum3(m+1,u));}
```

宏替换后，每一层递归调用中，将调用函数maxsum3超过2次

总结：代码调优

- 最重要的原理：尽量少用代码调优。
  - 不成熟的优化是大量编程灾难的根源，它会危及程序的正确性、功能性以及可维护性。
- 是否需要、以及 何时使用代码调优？
  - 效率很重要。
  - 遵循“没有坏的就不要修”。通过性能监视工具确定最耗时代码，并进行代码调优
    - 准则：多数的时间都消耗在少量的热点代码上。
  - 尽量从设计层面来提高效率。只有当没有更好的解决方案时才进行代码调优。
  - “双刃剑”。
    - 比如，将函数转化为宏，不一定会有加速效果。
    - “玩火者，小心自焚。”

- 代码调优的通用法则:(《编程珠玑》附录D)
  - 利用等价的代数表达式：比如，模运算优化。
  - 利用宏替换函数。
  - 使用哨兵来合并测试条件
  - 展开循环
  - 高速缓存需经常处理的数据。

设计层次	加速系数	修改
算法和数据结构	12	二叉树，将时间由 $O(n^2)$减少为 $O(n \log n)$
算法优化	2	使用更大的时间步
数据结构重组	2	产生适合树算法的簇
系统独立性代码优化	2	使用单精度浮点数替换双精度浮点数
系统依赖性代码优化	2.5	使用汇编语言重新编码关键函数
硬件	2	使用浮点加速器
总计	400	

《编程珠玑》6.1节中的加速情况汇总表

二分查找的边界情况

// Case1: while(l<=r) or
while(l<r) == return -1 or (a[l]==t)

```
#include<iostream>
using namespace std;
const int N=10;
int binarysearch(int *a, int t){
    int l=0,r=N-1;
    while (l<=r){
        int mid = (l+r) >>1;
        if(a[mid] > t){ r = mid-1;}//
        else if(a[mid] < t ) { l =
mid+1;}
        else return mid;
    }
    return -1; }
```

```
#include<iostream>
using namespace std;
const int N=10;
int binarysearch(int *a, int t){
    int l=0,r=N-1;
    while (l<r){
        int mid = (l+r) >>1;
        if(a[mid] > t){ r = mid-1;}
        else if(a[mid] < t ) { l =
mid+1;}
        else return mid;
    }
    return (a[l] == t)? l : -1; }
```

二分查找的边界情况

// Case2: while(l<r) r=mid, l=mid (?)

```
#include<iostream>
using namespace std;
const int N=10;
int binarysearch(int *a, int t){
    int l=0,r=N-1;
    while (l<r){
        int mid = (l+r) >>1;
        if(a[mid] > t){ r = mid-1;}
        else if(a[mid] < t ) { l = mid+1;}
        else return mid;
    }
        return (a[l] == t)? l : -1; }
```

```
#include<iostream>
using namespace std;
const int N=10;
int binarysearch(int *a, int t){
    int l=0,r=N-1;
    while (l<r){
        int mid = (l+r) >>1;
        if(a[mid] > t){ r = mid;}
        else if(a[mid] < t ) { l = mid+1;}
        else return mid;
    }
    return (a[l] == t)? l : -1; }
```

二分查找的边界情况

// Case3: Change N=10 to 15 the index of 2?

```
#include<iostream>
using namespace std;
const int N=11;
int binarysearch(int *a, int t){
    int l=0,r=N-1;
    while (l<r){
        int mid = (l+r) >>1;
        if(a[mid] > t){ r = mid-1;}
        else if(a[mid] < t ) { l = mid+1;}
        else return mid;
    }
        return (a[l] == t)? l : -1;
    }
```

```
int main()
{
    int a[N]={1,2,2,2,3,3,3,4,4,5};//N=10
    int a[N]={1,2,2,2,3,3,3,4,4,5,6};//N=11
    int a[N]={1,2,2,2,3,3,3,4,4,5,6,6,6,6,6}; //
N=15
    int r;
    r = binarysearch(a,2);
    cout<<"Target Index: "<<r<<endl;
}
```

例题：给定一个按照升序排列的长度为 n的整数数组，以及 q个查询。对于每个查询，返回一个元素 k的起始位置和终止位置（位置从 0 开始计数）。如果数组中不存在该元素，则返回 -1 -1。

输入格式
第一行包含整数 n和 q，表示数组长度和询问个数。第二行包含 n个整数（均在 1~10000 范围内），表示完整数组。接下来 q 行，每行包含一个整数 k，表示一个询问元素。1≤n≤100000　1≤q≤10000　1≤k≤10000

输出格式
共 q 行，每行包含两个整数，表示所求元素的起始位置和终止位置。如果数组中不存在该元素，则返回 -1 -1。

输入样例:
```
6 3
1 2 2 3 3 4
3
4
5
```

输出样例:
```
3 4
5 5
-1 -1
```

伪代码（左右边界模板）

算法 2.5　Binary Search for Range of Target Values

Input: $a[1 \cdots N]$, a sorted array of integers.

Input: q, the number of queries.

Input: tar, the target value for each query.

```
1  for each query do
2      Set l ← 0, r ← N − 1;
3      while l < r do
4          mid ← ⌊(l+r)/2⌋;
5          if a[mid] ⩾ tar then
6              r ← mid;
7          end
8          else
9              l ← mid + 1;
10         end
11     end
12     if a[l] = tar then
13         Output l (left index);
14         Set l ← 0, r ← N − 1;
15         while l < r do
16             mid ← ⌊(l+r+1)/2⌋;
17             if a[mid] ⩽ tar then
18                 l ← mid;
19             end
20             else
21                 r ← mid − 1;
22             end
23         end
24         Output l (right index);
25     end
26     else
27         Output −1, −1;
28     end
```

例题：给定一个浮点数 n，求它的三次方根。

输入格式：共一行，包含一个浮点数 n

输出格式：共一行，包含一个浮点数，表示问题的解。

注意，结果保留 6位小数。

数据范围
$-10000 \le n \le 10000$
输入样例：
1000.00
输出样例：
10.000000

C++代码

```
double q(double x){ return x*x*x;}

int main(){
    double x;
    cin >> x;

    double l = -10000, r = 10000;
    while(r-l>= 1e-7){
        double mid = (l+r)/2;
        if(q(mid) >= x) r = mid;
        else l = mid;
    }
    cout<<fixed<<setprecision(6)<<l;
    return 0;
}
```

思考

1、如何在程序中使用哨兵来找出数组中的最大元素。

提示：

```
i=0;
while i<n
{  max=x[i];
   x[n]=max;
   i++;
   while x[i]<max
            i++;       }
```

```
i=0;
max=x[i];
while i<n
{
    if x[i]>max
          max=x[i];
    i++; }
```

2、编程计算$y=a_nx^n+a_{n-1}x^{n-1}+\cdots\cdots+a_1x^1+a_0$

提示：

a_n

a_nx+a_{n-1}

$(a_nx+a_{n-1})x+a_{n-2}$

$((a_nx+a_{n-1})x+a_{n-2})x+a_{n-3}$

……

```
y=a[n];
for (i =n-1;i>=0;i--)
          y = x*y +a[i];
```

总结：顺序查找 vs. 二分查找

- 假设表长为n,则：
- s次顺序查找的时间开销：正比于s*n；
- s次二分查找的时间开销=对表排序的时间开销+s次二分查找的时间开销（正比于$s*\log_2 n$）；

例题（二分）

nums 为包含了n+1个元素的整数数组，其数值都在 [1, n] 范围内，可知至少存在一个重复的整数。假设 nums 只有 一个重复的整数 ，返回 这个重复的数 。

要求：不修改 数组 nums 且只用常量级 O(1) 的额外空间。

示例 1 :
输入：nums = [1,3,4,3,3]
输出：3

朴素方法

为1-N的每个元素打标记并记录出现的次数，记为tmp[i] 。
需要额外的存储空间

对于输入：

nums	4	1	3	3	3

index	0	1	2	3	4
tmp	0	1	0	3	1

直接强行省去这额外空间，则复杂度为O(N^2)，：N个数进行遍历

遍历tmp位置上不为1的数并输出, 即为所得结果。

时间/空间复杂度均为O(N)

如何优化？

二分查找

思路和算法

定义记录 cnt[i]，表示<=i的数的个数。我们举一个更为复杂的例子, 重复的数为 target = 5,则每个数字的对应cnt值如下：

nums	1	3	2	4	6	5	5	9	8	10	7
cnt	1	2	3	4	5	5	6	7	8	9	10

我们可以发现如下规律：[1, target -1] 里的所有数满足 cnt[i] ≤ i ；[target , n] 里的所有数满足 cnt[i] > i, 具有单调性

但这个性质一定是正确的吗？

单调性

nums 数组共有 n+1 个位置，其中数值均在 [1,n] 间，有且只有一个数重复了两次以上。

考虑以下两种情况：

- 数组中 target 出现了两次，其余的数各出现了一次，这个时候肯定满足上文提及的性质，因为小于 target 的数 i 满足 cnt[i] = i，大于等于 target 的数 j 满足 cnt[j] = j+1。
- 若 target 出现了三次及以上，必然有一些数被 target 替换。
    - 替换的数 i 小于 target，那么 [i, target - 1] 的 cnt 值均减一，其他不变，满足条件。
    - 替换的数 j 大于等于 target，那么 [target, j - 1] 的 cnt 值均加一，其他不变，亦满足条件。

```
int findDuplicate(vector<int>& nums) {
    int n = nums.size();
    int l = 1, r = n - 1, ans = -1;
    while (l <= r) {
        int mid = (l + r) >> 1;
        int cnt = 0;
        for (int i = 0; i < n; ++i) {
            cnt += nums[i] <= mid;}
        if (cnt <= mid) {
            l = mid + 1;
        } else {
            r = mid - 1;
            ans = mid;
        }
    }
    return ans;}
```

边界条件非常容易错，还是套用模板方便

```
int findDuplicate(vector<int>& nums) {
        int n = nums.size();
        int l =1, r = n-1;
        while (l < r) {
            int mid = (l + r) >> 1;
            int cnt = 0;
            for (int i = 0; i < n; ++i) {
                cnt += nums[i] <= mid;}
            if (cnt > mid) {
                r = mid;
            } else {
                l = mid + 1;
            }
        }
        return l;}
```

相当于找第一个cnt[i] > i的，也就是右半边的左起点

方法二：二进制

思路和算法

这个方法中我们将所有数二进制展开，按位考虑如何找出重复的数，如果我们能够确定重复数每一位是 1 还是 0 ，就可以按位还原出重复的数是什么。

考虑第 i 位，我们记 *nums* 数组中二进制展开后第 i 位为 1 的数有 x 个，数字 $[1, n]$ 这 n 个数二进制展开后第 i 位为 1 的数有 y 个，那么重复的数第 i 位为 1 当且仅当 $x > y$。

仍然以示例 1 为例，以下的表格列出了每个数字二进制下每一位是 1 还是 0 以及对应位的 x 和 y 是多少

	1	2	4	3	3	x	y
第 0 位	1	0	0	1	1	3	2
第 1 位	0	1	0	1	1	3	2
第 2 位	0	0	1	0	0	1	1

按之前所说，我们发现第 0 位和第 1 位的 $x > y$, 所以按位还原后 $target = (011)_2 = (3)_{10}$ ，符合答案

- 如果测试用例的数组中 *target* 出现了三次及以上，那么必然有一些数不在 *nums* 数组中，这个时候相当于我们用 *target* 去替换了这些数，我们考虑替换的时候对 x 的影响：

  - 如果被替换的数第 i 位为 1，且 *target* 第 i 位为 1：
    x 不变，满足 $x > y$
  - 如果被替换的数第 i 位为 0，且 *target* 第 i 位为 1：
    x 加一，满足 $x > y$
  - 如果被替换的数第 i 位为 1，且 *target* 第 i 位为 0：
    x 减一，满足 $x \leq y$
  - 如果被替换的数第 i 位为 0，且 *target* 第 i 位为 0：
    x 不变，满足 $x \leq y$

```
for (int bit = 0; bit <= bit_max; ++bit) {
    int x = 0, y = 0;
    for (int i = 0; i < n; ++i) {
        if (nums[i] & (1 << bit)) {
            x += 1;
        }
        if (i >= 1 && (i & (1 << bit))) {
            y += 1;}
    }
    if (x > y) {
        ans |= 1 << bit;
    }
```