优雅地写一个二分法

by Max.D.

二分是一项重要编程技能，在数学求解，方案可行性问题，查找问题等方面有着广泛应用。下面我们来学习如何优雅地设计二分法。

单调性方程求解

以本次上机Lx解方程为例

$$\frac{sin(\sqrt{x})+e^{-(x^{\frac{1}{3}})}}{ln(\pi x)}=y$$

$$x\in[0.33,10]$$

可以发现，在定义域之中，方程左边是一个单调函数，并且是递减的（几何画板or直观感受），于是可以用二分法解决。

我们可以不断猜测一个[L,R]中间的一个值M，根据图像，L的函数值在y上方，而R的函数值在y下方，那么如果M的函数值在y上方，显然根在右半区间（那个啥，叫零点存在定理？），否则在左半区间，这样，我们让L或者R成为新的M，重复以上的计算过程，这样区间[L,R]序列构成了一个闭区间套，理想上，能够套中一个唯一的根，实际实现上，我们迭代到所需要的精度就ok了，也可以用for循环到指定的次数。

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

double y,L=0.33,R=10.0,M;

int main()
{
    scanf("%lf",&y);
    while(R-L>1E-8)  //一般会把区间长度设置成比答案精度高几位，防止浮点误差
    {
        M=(R+L)/2;
        if((sin(sqrt(M))+exp(-pow(M,1.0/3)))/log(acos(-1)*M)>y)
        //别写成1/3了qwq，1/3=0
            L=M;
        else
            R=M;
    }
    printf("%.5lf",L);
    return 0;
}
```

这么做，效率O（logk）的。这里的k是原始范围和需求误差范围的比值大小。需要注意的是，精度的要求有时候是相对精度，比如要求的是你的答案与标准答案差的绝对值和标准答案的商小于一个极小数，这样也只需要改一下while里的条件即可

大家也可以思考一下，**为什么一般我们使用的是二分，而不是三分或者更多其它分？**

延伸：单峰函数求解极值

可以通过求导数，再二分出导数的零点，可得原函数极值点。

倘若不能求导，就有些麻烦了，有兴趣可以查阅“三分法”或者更优越的“0.618法”等方法。

整数：二分查找

考虑这样一种问题，原始有n大约三十万个数字，接着会有大量询问，查询某一个数字在其中是否存在。一个普遍的想法是数组存放，再从左到右逐一查找。但这样太低效了。

更好的方法是先将所有数字排序，之后用二分查找看看询问的数字是否存在。这么做对待每个询问，最差情况是O(logn)的，足够高效了。

为了规范性和通用性，下面的所有函数都用指针进行设计，不喜欢指针的同学完全可以用数组传参，直接求解数组下标。另外，递归也是设计二分的好方法，有兴趣的同学可以尝试

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

int a[300005],n,ask;

int cmp(const void *p, const void *q)
{
    return *((int *)p)>*((int *)q)?1:-1;
}

int binary_search(int *L, int *R, int val) //存在返回1，否则为0
{
    int *M;
    while(L<R)
    {
        M=L+(R-L>>1);
        if(*M<val)
            L=M+1;  //思考，为什么这里需要加一，同样，下面为什么减一
        else if(*M>val)
            R=M-1;
        else
            return 1; //一旦找到这个数字，直接返回
    }
    return *L==val; //范围缩小到唯一一个数时，检查这个数
}

int main()
{
    scanf("%d",&n);
    for(int i=1;i<=n;i++)
        scanf("%d",&a[i]);
    qsort(a+1,n,sizeof(int),cmp); //从小到大排序
    while(scanf("%d",&ask)==1)
        puts(binary_search(a+1,a+n,ask)?"Yes":"No");
    return 0;
}
```

这里定义了一个binary_search函数，检查数字存在性，参数分别为数列第一个数地址，最后一个数地址，以及询问数值。最差效率是O(logn)，当然如果一个数字频率特别大，二分的次数将会减少。如果想要得到一个数字的排名，将返回值修改为地址或者数组下标即可。

当然，有时问题不会这么简单。如果改成求一组数中，比某个数严格小的数字有多少个（注意不管询问是否在原数组中存在了），或者说这个数在原数列中首个插入而保持有序的位置，该怎么办呢？其实也不难

```
int* lower_bound(int *L, int *R, int val)
{
    int *M;
    while(L<R)
    {
        M=L+(R-L>>1);
        if(*M<val)
            L=M+1; //思考，这里为什么加一，而下面不加？注意我们要寻找的范围
        else
            R=M;
    }
    return L;
}
```

这个函数的功用，是返回有序数列中第一个大于或等于val的位置，或者说，找到了一个最左的位置，val在这个位置插入，而之后的数字后移一位，原数列将会保持有序。

另外非常重要的一点，在调用方面，R将不再是元素最后一个数的地址，而是元素最后一个数的后一位的地址(换句话说，是lower_bound(a+1,a+n+1,val);)，大家思考一下，为什么要这样？（想想如果val比所有数字大）

如果要求严格比val小数的个数，该怎么做呢？容易想到是lower_bound的答案的前一个位置-a。或者直接用二分，求最后一个严格比val小的数字的位置-a。代码应该是这样子。

```
int* upper_limit(int *L, int *R, int val)
{
    int *M;
    while(L<R)
    {
        M=L+(R-L+1>>1); //这里是个很有趣的地方，由于是R=M-1,所以加了个1，为的是防止死循环。思考为什么？
        if(*M<val)
            L=M;
        else
            R=M-1;
    }
    return L;
}
```

同样非常重要的一点，在调用方面，L将不再是元素第一个数的地址，而是元素第一个数的前一位的地址(换句话说，是upper_limit(a,a+n,val);)

相信大家可以根据以上函数，设计成自己需要的函数，比如找出有序数列中第一个严格大于val的位置（upper_bound），或者一个降序数组中的查找等等。篇幅原因不再累述。

二分答案求解可行性问题

例：有N个整数Xi， Xi值的范围从0到1000000000。要从中选出C个数（2<=C<=N），使得任意两个数差的绝对值的最小值尽可能大，求这个最大值。

容易想到，先将n个数组排序，那么我们要做的就是顺次选取。显然最小的数选入不会让答案变得更差劲。

直接完成这题相当困难，所幸我们有二分法。

对于猜测答案x，我们知道，如果能找到x对应的选取方案，那么真正答案不会比x小，反之一定比x小。

怎么寻找方案呢，这里用了贪婪法。以下是核心部分代码

```
L=0,R=1000000000;
while(L<R)
{
    M=L+R+1>>1;
    last=1;
    cnt=1;
    for(int i=2;i<=n;i++)
        if(f[i]-f[last]>=M)
        {
            cnt++;
            last=i;
        }
    if(cnt>=C)
        L=M;
    else
        R=M-1;
}
```