二分

二分
二分查找
二分答案
STL

二分查找

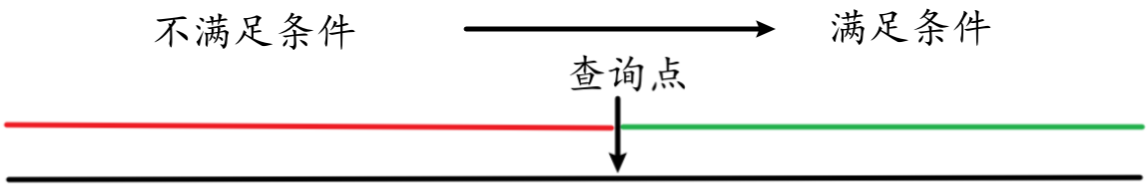
二分查找（binary search）：用来在一个有序数组中查找第一个/最后一个满足条件的元素的算法。

查找过程：

以在一个**升序**数组 a 中查找一个数为例。

在循环过程中，每次查询当前查找区间的中间元素，如果中间元素满足条件，就结束查找过程；如果中间元素小于所查找的元素，那么左侧的元素值只会更小，不可能存在所要查找的元素，因此只需要在右侧的区间内查找；如果中间元素大于所查找的元素，同理，只需要到左侧的区间内查找。

如图：



假设当前查找条件为：找到第一个大于等于 x 的值

红色部分的元素是不符合条件的（小于 x ），绿色部分是符合条件的（大于等于 x ）

那么对于这样一个数组，我们每次查询一个位置的元素，称当前查询的位置为查询点，则只有两种情况：

1.查询点落在红色部分

2.查询点落在绿色部分

并且在第一种情况中，查询点左侧的区间也为红色；在第二种情况中，查询点右侧的区间也为绿色

原因：假设当前查找的值为 x ，由于数组升序，则从左到右，元素的值会不断变大。

若查询点落在红色部分，则左侧的元素小于查询点位置的值，不符合条件（$<x$ ），

若查询点落在绿色部分，则右侧的元素大于等于查询点位置的值，符合条件（$\geq x$）

那么，我们可以确定要查找的值 x 不在红色部分，就可以缩小查找区间，继续查询。反复这一操作，直到满足退出循环的条件

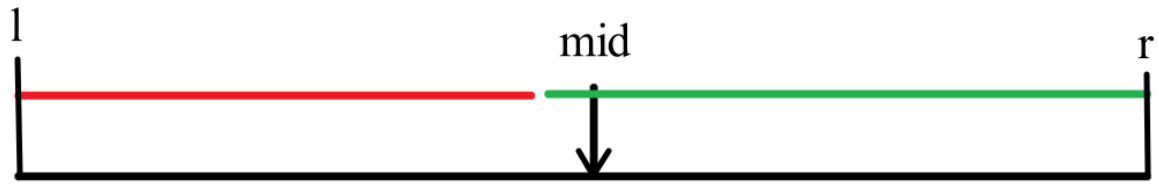
来看一个具体例子：**查找第一个满足下标对应的数大于等于给定数的下标**

解释：设升序数组为 a ，下标为 pos ，给定数为 x ，即要找到第一个 $a[pos] \geq x$ 的 pos 值。

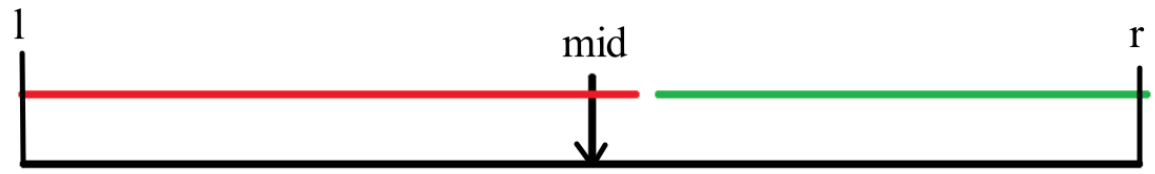
设查找区间为 $[l, r]$ ，每次查询当前区间的中点 mid ，检查是否满足条件： $a[mid] \geq x$

有两种情况：

1. $a[mid] \geq x$



2. $a[mid] < x$



对于第一种情况：

既然 mid 已经满足，它有可能是第一个满足条件的点，也有可能不是，所以应该将右端点 r 左移到 mid （$r = mid$）

对于第二种情况：

既然 mid 不满足情况，那么它左侧所有的点都不满足情况，所以应该将左端点 l 移动到 mid的右侧，即 $mid + 1$ （$l = mid + 1$）

重复以上步骤，直到 $l == r$，也就是说，当前区间只有一个数，那么答案就一定是 l

观察算法过程：在每一次循环中，我们会直接舍弃当前查找区间 $[l, r]$ 的一半，因此时间复杂度为 $O(log(r - l))$，自行证明，当做复杂度练习

那么，**查找第一个满足下标对应的数大于等于给定数的下标** 的二分查找代码如下：

```
void solve()
{
    int n,x;
    cin>>n>>x;
    vector<int> a(n);
    for(int i=0;i<n;i++)cin>>a[i];
    int l=0,r=n-1;//初始查找区间是数组长度
    while(l<r)
    {
        int mid=l+r>>1;//下取整找中点
        if(a[mid]>=x)r=mid;//如果当前都符合的话，那么右半边一定都符合，所以应该将右端点移动
到mid
        else l=mid+1;
    }
    cout<<l<<"\n";
}
```

除了找到第一个满足条件的元素，还有另一种情况：找到最后一个满足条件的元素

这种情况的图和上面的图正好相反。

上面的图是：不满足条件的红色部分在左侧，满足条件的绿色部分在右侧

这种情况的图是：满足条件的绿色部分在左侧，不满足条件的红色部分在右侧

来看具体例子：**查找最后一个满足下标对应的数小于等于给定数的下标**，过程大致相同，只是 l 和 r 的移动方式相反：

```
void solve()
{
    int n,x;
    cin>>n>>x;
    vector<int> a(n);
    for(int i=0;i<n;i++)cin>>a[i];
    int l=0,r=n-1;//初始查找区间是数组长度
    while(l<r)
    {
        int mid=l+r+1>>1;//上取整找中点
        if(a[mid]<=x)l=mid;//如果当前都符合的话，那么左半边一定都符合，所以应该将左端点移动到mid
        else r=mid-1;
    }
    cout<<l<<"\n";
}
```

对比一下两份代码，可以发现不同之处在于中点的选取和端点的移动.

第一份代码的中点是下取整 ($mid = l + r >> 1$)，第二份代码的中点是上取整 ($mid = l + r + 1 >> 1$)。原因是：如果第二份代码不写成上取整，则可能出现死循环。

问题在于左右端点的移动：假设当前 $l = 1, r = 2$，若中点是下取整，则 $mid = l + r >> 1$ 等于 1，$l = mid$ 仍为 1，陷入死循环

二分查找属于算法基础，不会有专门的题考二分查找，通常二分查找只是题目中的一部分

二分答案

解题的时候有时会考虑枚举答案，然后检验枚举的值是否正确/满足条件。若满足某种形式的单调性，则可以使用二分法枚举答案。利用二分的方法枚举答案，就是「二分答案」。

一般形式：

```
#include <bits/stdc++.h>
using namespace std;
#define ll long long

bool check(int x) // x为枚举的答案，check()函数作用为检验枚举的答案是否满足条件
{

}

void solve()
{
    /*
    一些代码
    */
```

```
    int l = 1, r = 1e9, mid; // 设置初始查找区间，端点的值结合具体题目，此处只是示例
    while (l < r)
    {
        /*
        端点移动代码，具体实现要结合题目
        */
    }
    //输出答案，如：cout<<l<<'\n';
}

int main()
{
    ios::sync_with_stdio(0);
    cin.tie(0);
    cout.tie(0);
    int T = 1;
    cin >> T;
    while (T--)
    {
        solve();
    }
    return 0;
}
```

例题：

COCI2011-2012#5] EKO / 砍树 - 洛谷 | 计算机科学教育新生态 (luogu.com.cn)

Problem - D - Codeforces

STL

C++ 标准库中实现了查找第一个不小于给定值的元素的函数 `std::lower_bound` 和查找第一个大于给定值的元素的函数 `std::upper_bound`，二者均定义于头文件 `<algorithm>` 中。

二者均采用二分实现，所以调用前必须保证元素有序。

```
auto it1 = lower_bound(a.begin(), a.end(), x);//找到数组a中第一个大于等于x的数的迭代器
auto it2 = upper_bound(a.begin(), a.end(), x);//找到数组a中第一个大于x的数的迭代器
```

这两个函数还有第四个参数：比较函数

```
bool cmp(int a, int b)//第二个参数b是我们给定的值，也就是x，最终会找到第一个不符合条件的数
{
    return a < b;//找到第一个大于等于b（即x）的数
}
```

注意：

对于 set 集合，如果对它直接使用 `lower_bound` 或 `upper_bound` ，复杂度为 $O(n)$，而不是 $O(logn)$。

应该使用其内置的 `lower_bound` 和 `upper_bound`，来达到 $O(logn)$ 复杂度

```
set<int>s;
int x=1;
auto it1=s.lower_bound(x);//同样返回迭代器
auto it2=s.upper_bound(x);
```

除了二分查找和二分答案，二分思想的拓展及应用有：三分，树上二分，整体二分和wqs二分

由于使用较少或难度较高，在此不做讲解