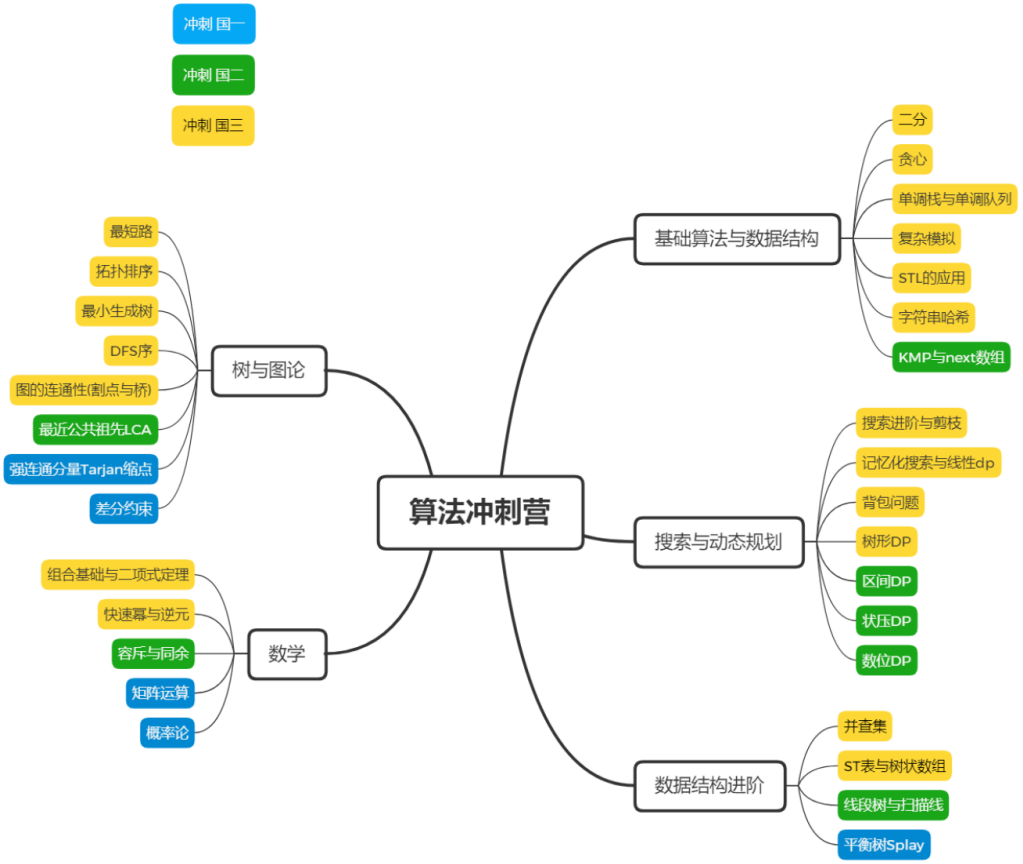
蓝桥杯30天算法冲刺集训



Day-1 二分

二分

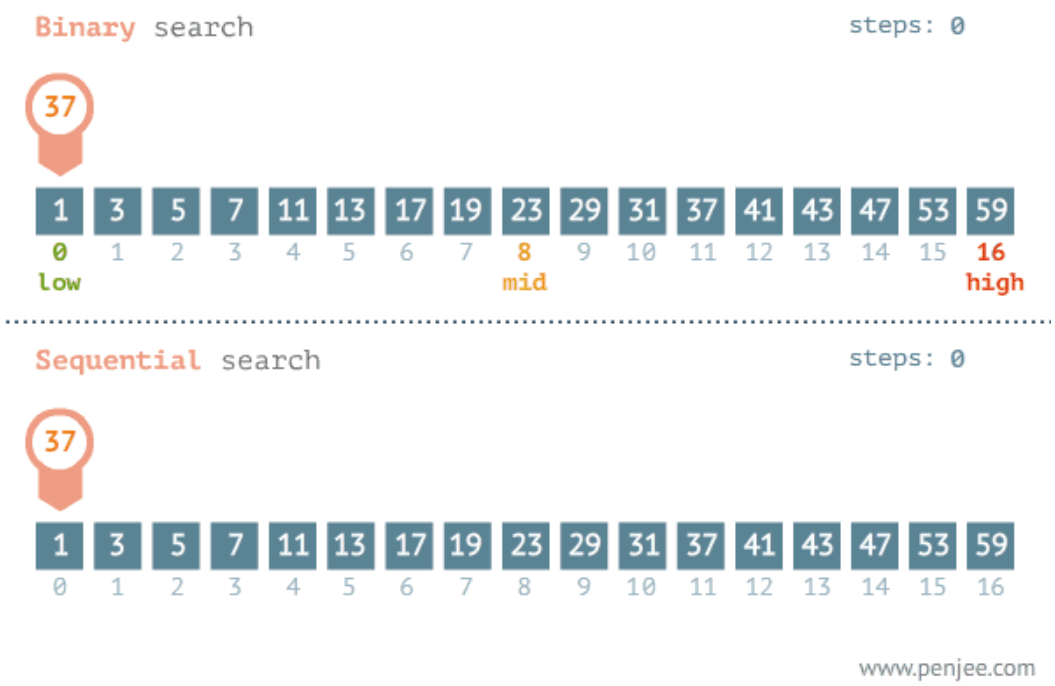
二分查找

二分查找算法思想：对于 n 个有序且没有重复的元素（假设为升序），从中查找特定的某个元素 x，我们可以将有序序列分为规模大致相等的两部分，然后取中间元素与要查找的元素 x 进行比较，如果 x 等于中间元素，则查找成功，算法终止；如果 x 小于中间元素，则在序列的前半部分继续查找，否则在序列的后半部分继续查找。这样就可以将查找的范围缩小一半，然后在剩余的一半中继续重复上面的方法进行查找。

这种每次都从中间元素开始比较，并且一次比较后就能把查找范围缩小一半的方法叫做二分查找。二分查找的时间复杂度是 $O(logn)$，是一种效率较高的查找算法。

一个比较形象的比喻就是大家翻书要翻到想看那一页，大家肯定都是先大概翻到中间

- 如果页码大了就直接往前翻，往后的页码都没必要翻了
- 如果页码小了就直接往后翻，往前的页码都没必要翻了



二分答案

二分思想不仅可以在有序序列中快速查找元素，还能高效地解决一些具有单调性判定的问题。

二分答案算法思想：某些问题不容易直接求解，但却很容易判断某个解是否可行，如果问题的答案具有单调性（即如果答案 x 不可行，那么大于 x 的解都不可行，而小于 x 的解都能可行），就像我们一开始玩的猜数字游戏一样，我们可以根据已知条件设定答案的上下界，然后用二分的方法枚举答案，再判断答案是否可行，根据判断的结果逐步缩小答案范围，直到符合题目条件的答案为止。

假设验证答案的时间复杂度为 $O(k)$，那么解决整个问题的时间复杂度为 $O(klogn)$。

lower_bound()和upper_bound()

lower_bound():二分查找左边界

```
lower_bound(a+开始,a+结束+1,x,cmp);
```

函数含义：在a数组的下标区间内，按照cmp的排序规则，找元素x的左边界(第一个 >=元素x的位置)，返回位置指针；(指针(Pointer): 变量的地址，通过它能找到以它为地址的内存单元。)

注意点：
(1)基本注意点同binary_search；
(2)此处返回的不是下标的值，而是返回指针；如果找不到符合条件的元素位置，指向下标为第一个大于元素的位置

例子：

```
#include<bits/stdc++.h>
using namespace std;
int main(){
    int a[6]={20,10,50,20,20,40};
    sort(a,a+5+1);//10 20 20 20 40 50
    int *p=lower_bound(a+0,a+5+1,20);
//  cout << p << " "<< *p << endl;
//  cout << p-a<< endl;
    cout << lower_bound(a,a+6,20)-a << endl;
    cout << lower_bound(a,a+6,15)-a << endl;
    cout << lower_bound(a,a+6,60)-a << endl;
    return 0;
}
```

upper_bound()：二分查找右边界

```
upper_bound(a+开始, a+结束+1,x,cmp);
```

函数含义：在a数组的下标内，按照cmp的排序规则，找元素x的 右边界(第一个 > 元素x的位置)，返回位置指针；
注意点：同 lower_bound；

例子：

```cpp
#include<bits/stdc++.h>
using namespace std;
int main(){
    int a[6]={20,10,50,20,20,40};
    sort(a,a+5+1);
    int *p=upper_bound(a,a+6,20);
    cout << p << " "<< *p << endl;
    cout << p-a<< endl;
    cout << upper_bound(a,a+6,20)-a << endl;
    cout << upper_bound(a,a+6,15)-a << endl;
    cout << upper_bound(a,a+6,60)-a << endl;
    return 0;
}
```

二分注意事项

1. 一定要注意能否二分的问题，二分都必须伴随着单调性的考虑，所谓单调性可以理解为能不能用二分，也就是对于二分最开始的$[l,r]$的区间内，是不是所有的每个单个元素都是满足所谓的转移的性质的
2. 一定要注意复杂度问题，如果有的二分需要排序，那么需要排序的二分是$O(NlogN)$的复杂度
3. 二分答案需要很强大的应用技巧，每个问题需要单独考虑怎么去做

蓝桥杯真题详解

质数变革（蓝桥杯C/C++2024B组第二场省赛）

这个题目是C/C++ B组2024第二场省赛的最后一题，虽然是最后一题但是难度并不算大，本题的要点在于二分的最基本的应用，也就是**找数**。所谓**找数**就是指我现在有一个已经排好序的序列，我从中**静态**的去找一个数的前驱和后继。静态指的是序列有序并且不会再进行改变。前驱指的是比这个数小的数，后继指的是比这个数大的数。因为都是数组上面去寻找所以复杂度都是$O(logN)$，其中N可以看成是符合范围内的素数的个数，每次找到下标后的偏移也都是$O(1)$。

本题的其他要点在于需要处理素数的筛选，用之前冲刺省一讲义里面的素数筛法即可实现，这里不再赘述。因为只需要找到10^6范围里面的素数，所以直接使用埃式筛法即可。

本题的第三个要点在于需要掌握一个叫做**调和级数**的东西，所谓调和级数也就是下面的：

```
for(int i=1;i<=n;i++)
    for(int j=i;j<=n;j+=i)
        // do something
```

需要注意的是这种代码的复杂度是$O(NlogN)$的。简要证明如下：

$$n+\frac{n}{2}+\frac{n}{3}+\ldots+\frac{n}{n}=n(1+\frac{1}{2}+\frac{1}{3}+\ldots+\frac{1}{n})$$

而下面这个这个式子的增长速度与lnN同阶，

$$n(1+\frac{1}{2}+\frac{1}{3}+\ldots+\frac{1}{n})$$

所以复杂度是$O(NlnN)=O(NlogN)$

这个东西就可以非常轻松的处理相关的**因数**与**倍数**之间的关系。

具体示例代码如下（筛选素数用的是欧拉筛，之前讲义里面也有提到过）：

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

const int N = 1e6 + 10; // 定义数组的最大大小

int a[N]; // 存储数字的数组
bool isPrime[N]; // 存储数字是否为素数的数组
int Prime[N], cnt = 0; // 存储素数和素数计数的数组
vector<int> G[N]; // 存储因子关系的向量数组
int dif[N], id[N]; // 存储差异和编号的数组

// 计算不大于n的素数
void GetPrime(int n)
```

```
{
    memset(isPrime, 1, sizeof(isPrime)); // 初始化所有数字为素数
    isPrime[1] = 0; // 1 不是素数

    for(int i = 2; i <= n; i++)
    {
        if(isPrime[i])
            Prime[++cnt] = i;

        for(int j = 1; j <= cnt && i * Prime[j] <= n; j++)
        {
            isPrime[i * Prime[j]] = 0;
            if(i % Prime[j] == 0)
                break;
        }
    }
}

// 对数字进行转换
int transform(int x, int y)
{
    int id = upper_bound(Prime + 1, Prime + 1 + cnt, x) -
Prime;
    if(y > 0)
    {
        --y;
        if(id + y > cnt)
            return 1;
        else
            return Prime[id + y];
    }
    if(y < 0)
    {
        if(id > 1 && Prime[id - 1] == x)
            --y;
        if(id + y < 1)
            return 0;
        else
            return Prime[id + y];
```

```
    }
}

int main()
{
    GetPrime(N - 1);
    --cnt;

    for(int i = 1; i < N; i++)
        for(int j = i; j < N; j += i)
            G[i].push_back(j);

    int n, m, op, k, x;
    cin >> n >> m;
    for(int i = 1; i <= n; i++)
        cin >> a[i];
    while(m--)
    {
        cin >> op >> k >> x;
        if(op == 1)
            dif[k] -= x;
        else
            dif[k] += x;
    }
    for(int i = 1; i < N; i++)
    {
        if(dif[i] != 0)
        {
            for(const auto it : G[i])
            {
                if(it >= 1 && it <= n)
                {
                    a[it] = transform(a[it], dif[i]);
                }
                else
                    break;
            }
        }
    }
```

```
    for(int i = 1; i <= n; i++)
        cout << a[i] << " ";
}
```

卡牌(蓝桥杯C/C++2022B组国赛)

这道题目是蓝桥杯B组国赛真题，可以当做一个练习二分 check 函数的一个好题。

首先我们要了解二分的上界与下界，在这个题目里面就是能够组成的套牌的数量，也就是最终的答案，并且可以发现这个答案是单调性的。

为何说是有**单调性**的呢？因为我们可以用当前的套牌以及手写的套牌两方面结合如果能组成N副套牌，则必然也能组成$N-1$副，$N-2$副...1副甚至0副套牌。

所以换句话说，**我们的跳转并不会出现问题**，那么直接二分就好了，但是还有个问题就是，我们如何去写相应的 check 函数呢？

实际上我们可以发现我们如果规定了当前的套牌的数量，那么我们是不是去选取套牌的方式就是一定的，也就是说我们需要从a_i里面选取，如果a_i不够还需要再手写当前i的套牌，那么整个 check 函数就算搞定了，只需要判断一下当前需要手写的套牌的数量是不是$\leq b_i$的即可。

最终我们算法的复杂度是二分的$(logN)$乘以 check 的$O(N)$，也就是$NlogN$级别的复杂度。

参考代码如下：

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

const long long inf = 0x3f3f3f3f; // 定义无穷大
const long long N = 2e5 + 5; // 定义数组大小
long long n, k; // 牌的数量和所需的空白牌数量
long long l, r; // 二分搜索的左右边界
long long ans; // 最终答案
long long p[N]; // 存放绘制每张牌所需的时间
long long m[N]; // 存放每张牌已有的空白牌数量
```

```cpp
// 检查是否满足条件
bool check(long long x)
{
    long long need = 0; // 存放所需的空白牌数量
    for (long long i = 1; i <= n; i++)
    {
        if (x - p[i] > m[i])
        {
            return false; // 所需的空白牌数量大于允许绘制的牌
        }
        need += max(x - p[i], 0ll); // 累计所需的空白牌数量
    }
    if (need <= k)
    {
        return true; // 空白牌数量足够
    }
    else
    {
        return false; // 空白牌数量不够
    }
}

int main()
{
    scanf("%lld%lld", &n, &k); // 输入牌的数量和所需的空白牌数量
    for (long long i = 1; i <= n; i++)
    {
        scanf("%lld", &p[i]); // 输入每张牌绘制所需的时间
    }
    for (long long i = 1; i <= n; i++)
    {
        scanf("%lld", &m[i]); // 输入每张牌已有的空白牌数量
    }
    l = 0; // 初始化左边界为0
    r = inf; // 初始化右边界为无穷大
    for (; l <= r;)
    {
        long long mid = (l + r) / 2; // 取中间值
```

```
        if (check(mid))
        {
            ans = mid; // 更新答案为当前中间值
            l = mid + 1; // 如果当前中间值满足条件，则尝试更大的值
        }
        else
        {
            r = mid - 1; // 如果当前中间值不满足条件，则尝试更小的
值
        }
    }
    printf("%lld\n", ans); // 输出最终答案
    return 0;
}
```

递增三元组(蓝桥杯C/C++2018B组省赛)

直接暴力枚举大家肯定都会，但是$O(N^3)$只有30%的分数，会直接超时。

我们考虑如何优化我们发现这个三元组要满足的条件是

$$A_i < B_j < C_k$$

可以看到我们如果先枚举两层for，然后再对B_j进行二分C_k，这样子也可以做，但是这样的复杂度是$O(N^2logN)$的，只能过60%的分数。

这种做法的具体如下伪代码所示：

```
for(int i=1;i<=n;i++)
    for(int j=1;j<=n;j++)
        if(a[i]<b[j])
            ans+=upper_bound(c+1,c+1+n,b[j])-c;
```

那么还是需要优化的，我们已经有了需要二分的思路了，就是怎么再把前面的两个for循环优化一下

我们再观察这个式子，发现其中的B_j是不是正好在中间那么是不是我们可以把上面的式子变换一下，如下所示：

$$\begin{cases} B_j > A_i \\ B_j < C_k \end{cases}$$

那么我们是不是直接枚举B_j去寻找两个不等式的个数然后乘起来就好了。

示例代码如下：

```
#include <bits/stdc++.h>
#define int long long
using namespace std;
const int N = 1e5 + 5;
int n , a[N] , b[N] , c[N] , ans;
signed main(){
    cin >> n;
    for(int i = 1; i <= n; i++) cin >> a[i];
    for(int i = 1; i <= n; i++) cin >> b[i];
    for(int i = 1; i <= n; i++) cin >> c[i];
    sort(a + 1 , a + 1 + n);
    sort(c + 1 , c + 1 + n);
    //排序，好进行二分
    for(int j = 1; j <= n; j++){
        int cnta = lower_bound(a + 1 , a + 1 + n , b[j]) - a -
1;
        int cntc = upper_bound(c + 1 , c + 1 + n , b[j]) - c;
        cntc = n - cntc + 1;
        //二分找出i的种类数和j的种类数
        ans += cnta * cntc;//乘法原理累计答案
    }
    cout << ans;
    return 0;
}
```

练习题

小蓝与数轴

小蓝与换装大赛

小蓝与捉迷藏