信息学奥赛笔记22

欧拉筛 | 结构体 | 排序

欧拉筛

在之前的课程当中，我们已经学习过了**埃氏筛法求质数**。虽然埃氏筛已经非常高效了，但是它仍然访问一个数可能超过`1`次，例如：在判断到4的时候，我们将$4*2=8$和$4*4=16$筛除，在判断到8的时候，我们又将$8*2=16$筛除，16这个数被筛去了2次。

欧拉筛则是对埃氏筛的一种改进，也被称之为线性筛，它可以保证，每一个合数都**只被筛去了一次。**

欧拉筛的代码实现如下：

动态数组版本

```cpp
#include <bits/stdc++.h>
using namespace std;
bool st[10000010];
int main() {
    long long n;
    cin >> n;
    vector<int> prime;    // 该数组用来记录质数
    for (int i = 2; i <= n; i++) {
        if (!st[i]) prime.push_back(i);
        for (int j = 0; j < prime.size() && i * prime[j] <= n; j++) {
            st[i * prime[j]] = 1;
            if (i % prime[j] == 0) break;
        }
    }
    for (int i = 0; i < prime.size(); i++) cout << prime[i] << " ";
    return 0;
}
```

静态数组版本

```cpp
#include <bits/stdc++.h>
using namespace std;
bool st[10000010];
int prime[1000000];  // 该数组用来记录质数
int main() {
    long long n, k = 0;  // 用一个指针k来表示目前插入到了第几个质数，k本身就是质数数组的长度
    cin >> n;
    for (int i = 2; i <= n; i++) {
        if (!st[i]) prime[k++] = i;    //prime[k] = i，然后为了能够保存下一个质数，这里需要用到和链表的插入相似的操作，对k++。
        for (int j = 0; j < k && i * prime[j] <= n; j++) {
            st[i * prime[j]] = 1;
            if (i % prime[j] == 0) break;
        }
```

```
    }
    for (int i = 0; i < k; i++) cout << prime[i] << " ";
    return 0;
}
```

> 时间复杂度$O(n)$，每个数最多只被访问1次。
>
> 空间复杂度$O(n)$，需要开辟长度n的数组来标记每个数是否被访问。

在代码中，最神奇的话是程序的第12行`if (i % prime[j] == 0) break;`。这也是比起埃氏筛多了的一句话，去掉这句话，变成埃氏筛，加上这一句话变成欧拉筛。咱们简短的证明一下这句话为什么可以保证每个数只被筛去一次。

如果$i\%prime[j] == 0$说明$prime[j]$是i的最小**质因数**，那之所以埃氏筛会导致一个合数被筛去两次，就是因为我们使用了一个数第二小甚至更多小的质因数去筛它。

这句话比较难以理解，我们来举一个例子：6的因数有$2,3$，这两个数都是6的质因数，之所以埃氏筛会导致合数被筛去两次，就是因为我们在访问到3的时候，也把$3*2=6$筛去了，但是当我们发现$6\%2==0$的时候，就说明2是6的最小质因数，我们就不该进行下去了，这样就可以保证，一个数最多只被访问一次，不会因为从一个数第二小的质因数再去访问到它。

用严格的数学归纳法角度来进行证明的话：

要证明一个合数x只会被筛掉一次，即标记$st[x]=1$一次。首先，对于$a=p_1b$，b只会筛掉a一次，因为从小到大枚举$prime[j]$，保证$b*prime[j]$递增，因此不可能遇到a两次。假设有其他的数筛掉a，即a被不等于b的一个数c筛掉了，其中$a=p_xc$
①若$c>b$，则$p_x<p1$，与p_1是a最小的质因数矛盾，假设不成立；
②若$c<b$，则$p_x>p_1$，这意味着p_1是c的质因数。那么c从小到大筛掉它的素数倍，在筛到p_1c时就$break$了，所以到不了a。

综上所述，每个数只会被筛去一次，外层的i是线性复杂度，总时间复杂度为线性的。

所以有$O(n)$的时间复杂度。

欧拉筛 

埃氏筛 

结构体

为什么要学结构体——上帝篇

在C++的语法中，我们学习了$int,double,char$等基础的，由C++官方提供给我们使用的数据类型。

但是在代码的编写过程当中，由C++给我们提供的数据类型已经无法满足我们的需求。

比如说，我们需要保存10个学生的姓名$(name)$，和他们的年龄(age)。为了满足一一对应关系，我们可以写一个姓名数组和年龄数组来存储，但是假如需要对这10个学生按照年龄从小到大排序呢？由于排序是只能对一个数组进行操作的，我们会发现，当我们把年龄数组按照从小到大排序后，这就不与学生的姓名保持一一对应关系了。那如果有一个数据类型，其中既能保存字符串类型，又能保存整数类型就

好了。

> 我乞求上帝给我一个能保存不同数据类型的数据类型，祂一脚把我踹到《信息学一本通》面前让我看结构体。

```
struct stu {
    stirng name;
    int age;
};
```

> 我恍然大悟谢谢上帝给我开悟，让我明白了原来C++还有这么厉害的操作。

我们可以创建一个结构体把C++给我们提供的标准数据类型，或者是**自定义的数据类型**打包放在一起，成为一个新的数据类型，这个数据类型是由我们自己命名的，比如说在刚刚的例子中，这个新的数据类型叫做stu，这个数据类型里包含了一个字符串类型的变量$name$，和一个整数类型的变量age。

那我们该如何使用这样的结构体来创建变量呢？

> 我又一次跪在上帝面前乞求上帝能够告诉我定义结构体变量的方式，祂一巴掌把我扇回《信息学一本通》的面前：

```
#include <iostream>
struct stu {
    stirng name;
    int age;
};
using namespace std;
int main() {
    struct stu a;
    return 0;
}
```

> 我又一次恍然大悟谢谢上帝给我开悟，这一次祂把《信息学一本通》赠送给我让我以后不要再来找祂了。

结构体的定义和使用

定义一个结构体的时候，相当于咱们写了一个函数，函数只要不被调用，就只是存在计算机里备用，结构体也一样，只要不是真真正正的创建变量，都相当于只是告诉计算机我要使用这个类型，真的来了吗？来也没来，如来；真的用了吗？真的用了吗？用也没用，**如用**。

所以，**定义结构体是不会占用变量的存储空间**的，计算机只是知道，我可以用这个类型了，但是还没开始用。

真正**把结构体变量定义出来是会占用存储空间**的，此时我们真的需要这个变量来进行操作，这样的思想，进阶一点我们也可以称之为叫做**面向对象**编程。

咱们把需要用到的东西，都**封装**在一个结构体（类）里，等真正需要用到它的时候在，再进行**对象实例化**。

用上述的例子来看，stu就是一个**抽象的类**，`struct stu a`的这个a就是一个**对象**，定义a的这个过程，就称之为**对象实例化**。

一个对象含有自身的**属性(Attribute)**和**方法(Method)**。那咱们目前这个阶段，只需要知道对象的**属性**即可。

还是例如上述的a，它有两种属性，分别是**姓名（name）**和**年龄（age）**。我们在访问一个对象的时候，只访问对象是不对的，需要直接去访问对象的属性。

比如说：

```cpp
#include <iostream>
struct stu {
    stirng name;
    int age;
};
using namespace std;
int main() {
    struct stu a;
    a.name = "原神"
    a.age = 3;
    return 0;
}
```

在上述代码中，变量a的$name$属性，我们设置为`"原神"`，age属性设置为`"3"`。

单独访问a就是错的，编译器会报错。

```cpp
#include <iostream>
struct stu {
    stirng name;
    int age;
};
using namespace std;
int main() {
    struct stu a;
    a.name = "原神"
    a.age = 3;
    cout << a << endl; //← ← ← ← ← ← ← ← 错错错错错错错错错错错错错错错错错错错错错错错错错错错错
    return 0;
}
```

要么输出$a.name$，要么输出$a.age$，输出这个整体，编译器直接**爆红**。

结构体的初始化

我们可以通过直接赋值，或者是输入的形式来对一个结构体变量进行初始化值，如上述的例子↑就是一个结构体变量的赋值型初始化，我们也可以直接cin。

```cpp
#include <iostream>
struct stu {
    stirng name;
    int age;
};
using namespace std;
int main() {
    struct stu a;
    cin >> a.name >> a.age;
    return 0;
}
```

> 通过查阅上帝送给我的《信息学一本通》我还发现了很多更有意思的东西。

我们可以在创建一个结构体的时候，直接给他初始化值。

例如：

```cpp
#include <iostream>
struct stu {
    stirng name;
    int age;
};
using namespace std;
int main() {
    struct stu a("原神", 3);
    return 0;
}
```

可以在定义的这个变量a的后方增加一个括号，就像是写了一个函数传值一样，把我们想给这个结构体变量赋的值直接写在里面，就像这里的`"原神"`，`3`，直接传进结构体内。

在后面的学习中，我们会知道，这是使用了结构体的**构造函数**。当然现阶段来说学习这个内容会有点过于超前。

有关结构体的更详细介绍，请听下回分解。这节课还是需求大家会读代码就行。

排序

排序$(Sorting)$，顾名思义，就是对一个杂乱无章的数据进行标准化处理的一个过程，一个有序的数据可以方便我们操作和使用，竞赛中对于排序的要求非常高。以至于我们需要学习十余种排序方法。虽然过程不同，但是排序的结果相同，都是把一组数据变得有序。

> 我又一次来到上帝面前，刚想张口，低头翻开了《信息学一本通》，发现上帝赋予了我一个内置的排序函数，我开心的像个孩子一样笑了。

在C++的$< algorithm >$库中，有一个叫做$sort$的函数，顾名思义，它的作用就是给数组进行排序，将数组的值按照从小到大的顺序排列好。

静态数组的排序方式

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

```cpp
int a[10010], b[10010];
int main() {
    int n;
    cin >> n;
    for (int i = 0; i < n; i++) {
        cin >> a[i];
    }
    sort(a + 0, a + n);  // 对0 - (n - 1)型静态数组进行排序

    for (int i = 1; i <= n; i++) {
        cin >> b[i];
    }
    sort(b + 1, b + n + 1); // 对 1 - n 型静态数组进行排序
    return 0;
}
```

动态数组的排序方式

```cpp
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;
int main() {
    int n;
    cin >> n;
    vector<int> a(n), b(n + 1)
    for (int i = 0; i < n; i++) {
        cin >> a[i];
    }
    sort(a.begin(), a.end());  // 对0 - (n - 1)型动态数组进行排序

    for (int i = 1; i <= n; i++) {
        cin >> b[i];
    }
    sort(b.begin() + 1, b.end()); // 对 1 - n 型动态数组进行排序
    return 0;
}
```

经过这么一句话后数组立马就有序了，不信你可以试试。

不对就是你写错了！

C++内省排序(知识点拓展)

我们使用的$sort$函数其实并不是一种排序算法，而是由多种排序算法组成的一个结合体，这就比较像是**手动挡汽车**，在发动机转速达到一定时，为了提高汽车的速度，我们需要使用**挂档器**来对发动机的转速进行调整，从而达到提高发动机的转速的一个目的。而C++的内省排序正是沿用了这一原理。

设待排序的数据量为n。

当$n \leq 16$时，使用**插入排序**，较为稳定，最坏时间复杂度为$\mathrm{O}(n^2)$。

当$n > 16$时，优先**快速排序**，快速排序是一种**递归与分治**的算法。当递归深度过深，超过$2logN$时。自动切换为**堆排序**。

这样无论在任何规模的数据，任何情况下，都可以保证排序的**最坏时间复杂度**为$O(nlogn)$。保证了算法最优性原则。

总结一下就是：

- 当数据规模过小时，采用插入排序进行排序
- 当快速排序的栈深度过于深，可能已经陷入最差情况，转而采用堆排序
- 正常情况采用快速排序

你学会了吗?

你学会了吗?

你学会了吗?