题解：求n!中素因子p的个数

思路1——复杂度：1,440,329,197次乘除法+溢出

1.1. 想法和实现

> 首先计算出来n!，然后一直除以p，直到除干净为止。

这个思路的实现很简单，我们用 `num_p` 记录答案，如下:

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

int main() {
    int n;
    cin >> n;
    int factorial_n = 1;
    for (int i = 1; i <= n; ++i) {
        factorial_n *= i;
    }
    int num_p = 0;
    while (factorial_n % p == 0) {
        num_p += 1;
        factorial_n /= p;
    }
    cout << num_p;
    return 0;
}
```

但是这个方法有几个比较严重的问题:

- 首先，数值会溢出。阶乘函数增长得太快了。假如n=100，大家想一想100!有多大，大概是$9 * 10^{157}$，long long也存不下这么大的一个整数。
  而我们的测试机里面甚至有 `n=1234567890，p=13` 。
- 其次，这个方法复杂度太高，我们来估计一下要循环几次:
  - 计算阶乘要循环n次
  - 用n!不断除以p，循环次数就是 `num_p` 的值。粗略估计一下，最坏情况要循环$\log_p n!$次，由大家高数学过的斯特林公式可以知道，这个东西的主项大约是$n \log_p n$

所以总的循环次数的一个上界就是$n + n\log_p n$次。我们代入 `n=1234567890，p=13` ，循环次数的上界是 `11,310,567,211` 次。

但是实际上呢并没有这么多，因为n!不会刚好是p的幂。当 `n=1234567890，p=13` ， `num_p` 其实只有 `102,880,653` ，比 `n` 都还要小。

1.2. 计算开销分析

我们来算一下乘除运算的总次数（乘除开销是加减法的好几倍，就不考虑加减法了）：

- 在第一个循环里面，我们每循环一次做一次乘法
- 在第二个while循环里面，每循环一次做两次除法（ `/=p` 和条件里面的 `%p` ），还有一次最后的条件判断退出循环

所以总的乘除运算次数为 `n + 2 * num_p + 1 = 1,440,329,197` 。这大概是我们估计的上界的十分之一不到，但是也足够你超时了。

思路2——复杂度：300,728,066次乘除法

2.1. 想法和实现

我们换个角度来想:

> n!是从1到n所有数乘起来，那么n!里面的素因子p的个数就是由[1, n]中每个乘数贡献的。所以我们从1到n循环每个数，看看每个数中包含多少个p，再累加就可以。

实现如下：

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

int main()
{
        int n, p;
        cin >> n >> p;
        int num_p = 0;
        for (int i = 1; i <= n; i++) {
                int ii = i;
                int cnt = 0;
                while (ii % p == 0) {
                        cnt += 1;
                        ii /= p;
                }
                num_p += cnt;
        }
        cout << num_p;
    return 0;
}
```

2.2. 计算开销分析

现在我们并没有真正把n!算出来，不会有溢出的问题，但我们来算算复杂度呢？

- 外层循环看上去只循环了n次，但是内层还有循环。事实上，内层每循环一次，答案加一。所以内层循环次数就是 `num_p` 。

循环次数好像比思路一少了一个 `n` 诶，那这样做是不是会快一些呢？那来让我们精细地算一下乘除法的次数:

- 每次内层循环我们做两次除法操作（一次进入循环的判断，一次循环体里面的 `/=p` ）所以有 `2 * num_p` 次除法。
- 但是！对于每个数 `i` 还有一次额外的退出循环的判断，这里又有 `n` 次除法。

我们之前算过， `num_p = 102,880,653` ，所以现在总的乘除运算次数为 `n + 2 * num_p = 1,440,329,196` 次，其实只比之前的方案少一次！所以这样做仍然会超时。

2.3. 改进一下

我们还可以继续优化:

> 外层循环可以只考虑p的倍数

实现如下:

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

int main()
{
        int n, p;
        cin >> n >> p;
        int num_p = 0;
        for (int i = p; i <= n; i += p) {
                int ii = i;
                int cnt = 0;
                while (ii % p == 0) {
                        cnt += 1;
                        ii /= p;
                }
                num_p += cnt;
        }
        cout << num_p;
    return 0;
}
```

2.4. 计算开销分析

这样的话我们又节省掉一些退出内层循环的判断开销。那些不是p的倍数的数被跳过了，只有p的倍数才会贡献那一次额外的退出开销。所以现在总的乘除运算次数为 `n / p + 2 * num_p = 300,728,066` 减小到了原来的大约$\frac{1}{5}$。大家试一试，现在可以刚好过掉评测。

思路3——复杂度：32次乘除法

3.1. 想法和实现

让我们再来分析一下这个问题，我们是不是真的需要从1到n逐个判断每个数贡献了多少个p呢?

事实上，只有那些形如$r \cdot p^k$的数才会贡献k个 `p` 到 `n!` 里面去。对于每个指数k，我们可以很方便地计算$[1, n]$中有多少个形如$r \cdot p^k$的数，将它记为 `num_p_k` 的话，我们有 `num_p_k = n / p^k - n / p^(k+1)` 。

这就有了第三个解法，循环指数 k：

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

int main () {
        int n, p;
        cin >> n >> p;
        int num_p = 0;
        int base = p;
        for (int k = 1; k <= n; ++k) {
                num_p += (n / base - n / (base *= p)) * k;
                if (base > n) {
                        break;
                }
        }
        cout << num_p;
}
```

在这个程序里，我们用 `base` 来滚动保存p^k。

3.2. 计算开销分析

来让我们分析一下这个程序的复杂度吧，由于 `p^k > n` 的时候我们就会break，所以我们只循环了$\lfloor \log_p n \rfloor$次。

再来精细地分析一下乘除法次数，每次循环会做4次乘除法。所以当 `n = 1234567890, p = 13` 的时候，总的乘除法次数就是$4 * \log_p n = 32$次，是前面两种方法的**千万分之一**!

3.3. 一份混乱的例程

之前我在课堂上演示的写法是我瞎写的，没有上面这一份代码这么清晰，我也贴出来一下:

```cpp
#include <iostream>
using namespace std;
int main()
{
    int n, p;
    cin >> n >> p;
    //计算最大的power，使得p^power <= n
    //算完之后，base = p^power
    int power = 0;
    int base = 1;
    int nn = n;
    while (nn >= p) {
        power += 1;
        base *= p;
        nn /= p;
    }
    int cnt = 0;
    int num_p = 0;
    // 从power开始，从大到小循环p的指数i
    // [1, n]中有多少个p^i的倍数，结果就要加多少个i
    // 但是p^(i+1), p^(i+2), ... 都是p^i的倍数，不能重复统计它们
    // [1, n]中，恰好是p^i的倍数，而不是更高次幂的倍数的数究竟有多少个呢
    // 答案很简单，就是n/p^i - n/p^(i+1)，后者在循环里用一个cnt来记录
    for (int i = power; i >= 1; i--) {
        num_p += (n / base - cnt) * i;
        cnt = n / base;
        base /= p;
    }
    cout << num_p;
    return 0;
}
```

这份代码我先去计算了$\lfloor \log_p n \rfloor$的值，存储在 `power` 里面，然后倒着计算$\frac{n}{p^i} - \frac{n}{p^{i+1}}$。写麻烦了，不过我们也来分析一下复杂度吧！

首先计算 `power` ，循环了$\lfloor \log_p n \rfloor = 8$次，每次循环进行了两次乘除运算，所以一共是16次。下面同样循环了$\lfloor \log_p n \rfloor = 8$次，
每次里面有4次乘除运算，所以一共有32次，整个程序总共48次乘除计算。

4. 总结

这道题我觉得特别好，同学们可以感受到用循环来解决问题的时候**优化**的必要性。在做优化的时候，要注意这么几个问题：

- 想好循环的空间是什么（比如，是遍历数本身？还是遍历指数？）
- 思考优化的有效性，避免想当然（比如，思路二的第一个实现，看上去好像减少了循环次数，但其实和第一个思路几乎一样）
- 建立数学逻辑和计算机硬件特性的联系（比如，我们在程序设计的时候，要考虑到数值溢出/数值精度问题，要考虑到乘除法开销大于加减法，等等）