讲两道常考的阶乘算法题

Stars 107k | B站 @labuladong | 配套PDF和插件 下载 | 打卡挑战 报名 | 精品课程 查看

微信搜一搜 labuladong公众号

通知：数据结构精品课 V1.6 持续更新中，第八期打卡挑战 开始报名。

读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

牛客	LeetCode	力扣	难度
-	172. Factorial Trailing Zeroes	172. 阶乘后的零	🟠
-	793. Preimage Size of Factorial Zeroes Function	793. 阶乘函数后 K 个零	🔴

————

笔试题中经常看到阶乘相关的题目，今天说两个最常见的题目：

1、输入一个非负整数 `n`，请你计算阶乘 `n!` 的结果末尾有几个 0。

这也是力扣第 172 题「阶乘后的零」，比如说输入 `n = 5`，算法返回 1，因为 `5! = 120`，末尾有一个 0。

函数签名如下：

```
int trailingZeroes(int n);
```

2、输入一个非负整数 `K`，请你计算有多少个 `n`，满足 `n!` 的结果末尾恰好有 `K` 个 0。

这也是力扣第 793 题「阶乘后 K 个零」，比如说输入 `K = 1`，算法返回 5，因为 `5!,6!,7!,8!,9!` 这 5 个阶乘的结果最后只有一个 0，即有 5 个 `n` 满足条件。

函数签名如下：

```
int preimageSizeFZF(int K);
```

我把这两个题放在一起，肯定是因为它们有共性，下面我们来逐一分析。

题目一

肯定不可能真去把 `n!` 的结果算出来，阶乘增长可是比指数增长都恐怖，趁早死了这条心吧。

那么，结果的末尾的 0 从哪里来的？我们有没有投机取巧的方法计算出来？

首先，两个数相乘结果末尾有 0，一定是因为两个数中有因子 2 和 5，因为 10 = 2 x 5。

也就是说，问题转化为：`n!` 最多可以分解出多少个因子 2 和 5？

比如说 `n = 25`，那么 `25!` 最多可以分解出几个 2 和 5 相乘？这个主要取决于能分解出几个因子 5，因为每个偶数都能分解出因子 2，因子 2 肯定比因子 5 多得多。

`25!` 中 5 可以提供一个，10 可以提供一个，15 可以提供一个，20 可以提供一个，25 可以提供两个，总共有 6 个因子 5，所以 `25!` 的结果末尾就有 6 个 0。

现在，问题转化为：`n!` 最多可以分解出多少个因子 5？

难点在于像 25，50，125 这样的数，可以提供不止一个因子 5，怎么才能不漏掉呢？

这样，我们假设 `n = 125`，来算一算 `125!` 的结果末尾有几个 0：

首先，125 / 5 = 25，这一步就是计算有多少个像 5，15，20，25 这些 5 的倍数，它们一定可以提供一个因子 5。

但是，这些足够吗？刚才说了，像 25，50，75 这些 25 的倍数，可以提供两个因子 5，那么我们再计算出 `125!` 中有 125 / 25 = 5 个 25 的倍数，它们每人可以额外再提供一个因子 5。

够了吗？我们发现 125 = 5 x 5 x 5，像 125，250 这些 125 的倍数，可以提供 3 个因子 5，那么我们还得再计算出 `125!` 中有 125 / 125 = 1 个 125 的倍数，它还可以额外再提供一个因子 5。

这下应该够了，`125!` 最多可以分解出 25 + 5 + 1 = 31 个因子 5，也就是说阶乘结果的末尾有 31 个 0。

理解了这个思路，就可以理解解法代码了：

```
int trailingZeroes(int n) {
    int res = 0;
    long divisor = 5;
    while (divisor <= n) {
        res += n / divisor;
        divisor *= 5;
    }
    return res;
}
```

这里 `divisor` 变量使用 long 型，因为假如 `n` 比较大，考虑 while 循环的结束条件，`divisor` 可能出现整型溢出。

上述代码可以改写地更简单一些：

```
int trailingZeroes(int n) {
    int res = 0;
    for (int d = n; d / 5 > 0; d = d / 5) {
        res += d / 5;
    }
    return res;
}
```

这样，这道题就解决了，时间复杂度是底数为 5 的对数，也就是 `O(logN)`，我们看看下如何基于这道题的解法完成下一道题目。

第二题

现在是给你一个非负整数 `K`，问你有多少个 `n`，使得 `n!` 结果末尾有 `K` 个 0。

一个直观地暴力解法就是穷举呗，因为随着 `n` 的增加，`n!` 肯定是递增的，`trailingZeroes(n!)`

肯定也是递增的，伪码逻辑如下：

```
int res = 0;
for (int n = 0; n < +inf; n++) {
    if (trailingZeroes(n) < K) {
        continue;
    }
    if (trailingZeroes(n) > K) {
        break;
    }
    if (trailingZeroes(n) == K) {
        res++;
    }
}
return res;
```

前文 二分查找如何运用 说过，**对于这种具有单调性的函数，用 for 循环遍历，可以用二分查找进行降维打击**，对吧？

搜索有多少个 `n` 满足 `trailingZeroes(n) == K`，其实就是在问，**满足条件的 `n` 最小是多少，最大是多少，最大值和最小值一减，就可以算出来有多少个 `n` 满足条件了**，对吧？那不就是 二分查找 中「搜索左侧边界」和「搜索右侧边界」这两个事儿嘛？

先不急写代码，因为二分查找需要给一个搜索区间，也就是上界和下界，上述伪码中 `n` 的下界显然是 0，但上界是 `+inf`，这个正无穷应该如何表示出来呢？

首先，数学上的正无穷肯定是无法编程表示出来的，我们一般的方法是用一个非常大的值，大到这个值一定不会被取到。比如说 int 类型的最大值 `INT_MAX`（2^31 - 1，大约 31 亿），还不够的话就 long 类型的最大值 `LONG_MAX`（2^63 - 1，这个值就大到离谱了）。

那么我怎么知道需要多大才能「一定不会被取到」呢？**这就需要认真读题，看看题目给的数据范围有多大**。

这道题目实际上给了限制，`K` 是在 `[0, 10^9]` 区间内的整数，也就是说，`trailingZeroes(n)` 的结果最多可以达到 `10^9`。

然后我们可以反推，当 `trailingZeroes(n)` 结果为 `10^9` 时，`n` 为多少？这个不需要你精确计算出来，你只要找到一个数 `hi`，使得 `trailingZeroes(hi)` 比 `10^9` 大，就可以把 `hi` 当做正无穷，作为搜索区间的上界。

刚才说了，`trailingZeroes` 函数是单调函数，那我们就可以猜，先算一下 `trailingZeroes(INT_MAX)` 的结果，比 `10^9` 小一些，那再用 `LONG_MAX` 算一下，远超 `10^9` 了，所以 `LONG_MAX` 可以作为搜索的上界。

注意为了避免整型溢出的问题，`trailingZeroes` 函数需要把所有数据类型改成 long：

```
// 逻辑不变，数据类型全部改成 long
long trailingZeroes(long n) {
    long res = 0;
    for (long d = n; d / 5 > 0; d = d / 5) {
        res += d / 5;
    }
    return res;
}
```

现在就明确了问题：

在区间 `[0, LONG_MAX]` 中寻找满足 `trailingZeroes(n) == K` 的左侧边界和右侧边界。

根据前文 二分查找算法框架，可以直接把搜索左侧边界和右侧边界的框架 copy 过来：

```
/* 主函数 */
public int preimageSizeFZF(int K) {
    // 左边界和右边界之差 + 1 就是答案
    return (int)(right_bound(K) - left_bound(K) + 1);
}

/* 搜索 trailingZeroes(n) == K 的左侧边界 */
long left_bound(int target) {
    long lo = 0, hi = Long.MAX_VALUE;
    while (lo < hi) {
        long mid = lo + (hi - lo) / 2;
        if (trailingZeroes(mid) < target) {
            lo = mid + 1;
        } else if (trailingZeroes(mid) > target) {
            hi = mid;
        } else {
            hi = mid;
        }
    }
    return lo;
```

```
    }

    /* 搜索 trailingZeroes(n) == K 的右侧边界 */
    long right_bound(int target) {
        long lo = 0, hi = Long.MAX_VALUE;
        while (lo < hi) {
            long mid = lo + (hi - lo) / 2;
            if (trailingZeroes(mid) < target) {
                lo = mid + 1;
            } else if (trailingZeroes(mid) > target) {
                hi = mid;
            } else {
                lo = mid + 1;
            }
        }
        return lo - 1;
    }
```

如果对二分查找的框架有任何疑问，建议好好复习一下前文 二分查找算法框架，这里就不展开了。

现在，这道题基本上就解决了，我们来分析一下它的时间复杂度吧。

时间复杂度主要是二分搜索，从数值上来说 `LONG_MAX` 是 2^63 - 1，大得离谱，但是二分搜索是对数级的复杂度，log(LONG_MAX) 是一个常数；每次二分的时候都会调用一次 `trailingZeroes` 函数，复杂度 O(logK)；所以总体的时间复杂度就是 O(logK)。

_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _