

# 时间复杂度低于 $O(n)$ 的算法

主讲人 令狐冲  
课程版本 v7.0

# 除了二分法以外还有如下几种

快速幂算法  $O(\log N)$

辗转相除法  $O(\log N)$

分解质因数  $O(\sqrt{N})$

分块检索法  $O(\sqrt{N})$

# 快速幂算法

求  $a^n \% b$

其中  $a, b, n$  都是 int 范围 ( $2^{31} - 1$ )

```
class Solution {
    /*
     * @param a, b, n: 32bit integers
     * @return: An integer
     */
    public int fastPower(int a, int b, int n) {
        if (n == 1) {
            return a % b;
        }
        if (n == 0) {
            return 1 % b;
        }

        long product = fastPower(a, b, n / 2);
        product = (product * product) % b;
        if (n % 2 == 1) {
            product = (product * a) % b;
        }
        return (int) product;
    }
};
```

```
class Solution:
    """
    @param a: A 32bit integer
    @param b: A 32bit integer
    @param n: A 32bit integer
    @return: An integer
    """
    def fastPower(self, a, b, n):
        if n == 0:
            return 1 % b

        if n == 1:
            return a % b

        #  $a^n = (a^{n/2})^2$ 
        power = self.fastPower(a, b, n // 2)
        power = (power * power) % b

        # 如果 n 是奇数, 还需要多乘以一个 a, 因为 n // 2 是整除
        if n % 2 == 1:
            power = (power * a) % b

        return power
```

## 二进制的做法 —— 非递归，比较巧妙

```
public int fastPower(int a, int b, int n) {  
    long ans = 1, tmp = a;  
  
    while (n != 0) {  
        if (n % 2 == 1) {  
            ans = (ans * tmp) % b;  
        }  
        tmp = (tmp * tmp) % b;  
        n = n / 2;  
    }  
  
    return (int) ans % b;  
}
```

```
def fastPower(self, a, b, n):  
    ans = 1  
    while n > 0:  
        if n % 2 == 1:  
            ans = (ans * a) % b  
        a = a * a % b  
        n = n // 2  
    return ans % b
```

比如  $n=5$ , 可以看做  $a^{(101)_2} \% b$  (5的二进制是101)

拆开也就是  $a^{(100)_2} * a^{(1)_2} \% b$

$$a^{(1010)_2} = a^{(1000)_2} * a^{(10)_2}$$

因此相当于我们把  $n$  做二进制转换，碰到 1 的时候，称一下对应的  $a$  的幂次

而  $a$  的幂次我们只需要知道  $a^1, a^{(10)_2}, a^{(100)_2} \dots$  也就是  $a^1, a^2, a^4 \dots$

因此不断的把  $a = a * a$  就可以了

中间计算的时候，随时可以  $\% b$  避免 overflow 其不影响结果，这是  $\%$  运算的特性。

# 分块检索算法

将长度为  $N$  的区间分成  $\sqrt{N}$  的大小的小区  
总共  $\sqrt{N}$  个小区间，每个小区间统计局部的数据  
因此在这些区间中进行增删查改的效率是  $O(\sqrt{N})$

# 统计每个数前面比他小的数

<https://www.lintcode.com/problem/count-of-smaller-number-before-itself/>

<https://www.jiuzhang.com/solution/count-of-smaller-number-before-itself/>

[1, 2, 7, 8, 5] 每个数前面比他小的数分别为 [0, 1, 2, 3, 2]

```
class BlockArray:
    def __init__(self, max_value):
        self.blocks = [
            Block()
            for _ in range(max_value // 100 + 1)
        ]

    def count_smaller(self, value):
        count = 0
        block_index = value // 100
        for i in range(block_index):
            count += self.blocks[i].total

        counter = self.blocks[block_index].counter
        for val in counter:
            if val < value:
                count += counter[val]
        return count

    def insert(self, value):
        block_index = value // 100
        block = self.blocks[block_index]
        block.total += 1
        block.counter[value] = block.counter.get(value, 0) + 1
```

```
class Block:
    def __init__(self):
        self.total = 0
        self.counter = {}
```

```
class Solution:
    """
    @param A: an integer array
    @return: A list of integers includes the index of the smaller numbers
    """
    def countOfSmallerNumberII(self, A):
        if not A:
            return []

        block_array = BlockArray(10000)
        results = []
        for a in A:
            count = block_array.count_smaller(a)
            results.append(count)
            block_array.insert(a)
        return results
```



```
class BlockArray {
    public Block[] blocks;
    public int blockSize;

    public BlockArray(int capacity) {
        blockSize = (int) Math.sqrt(capacity);
        int blockCount = capacity / blockSize + 1;
        blocks = new Block[blockCount];
        for (int i = 0; i < blockCount; i++) {
            blocks[i] = new Block(blockSize);
        }
    }

    public int countSmaller(int value) {
        int index = value / blockSize;
        int count = 0;
        for (int i = 0; i < index; i++) {
            count += blocks[i].total;
        }

        for (int i = 0; i + index * blockSize < value; i++) {
            count += blocks[index].counter[i];
        }

        return count;
    }

    public void insert(int value) {
        int index = value / blockSize;
        blocks[index].total++;
        blocks[index].counter[value - index * blockSize]++;
    }
}
```

```
class Block {
    public int total;
    public int[] counter;
    public Block(int blockSize) {
        this.total = 0;
        this.counter = new int[blockSize];
    }
}
```

```
public class Solution {
    /**
     * @param A: an integer array
     * @return: A list of integers includes the index of t
     */
    public List<Integer> countOfSmallerNumberII(int[] A) {
        List<Integer> results = new ArrayList<>();
        if (A == null || A.length == 0) {
            return results;
        }

        BlockArray blockArray = new BlockArray(10000);
        for (int i = 0; i < A.length; i++) {
            results.add(blockArray.countSmaller(A[i]));
            blockArray.insert(A[i]);
        }

        return results;
    }
}
```