a	n(binary)	ans
$7^{(0)_2}$	1010	1
$7^{(10)_2}$	101	$7^{(10)_2}$
$7^{(100)_2}$	10	$7^{(10)_2}$
$7^{(1000)_2}$	1	$7^{(10)_2} \cdot 7^{(1000)_2}$

算法学习笔记(4)：快速幂

Pecco
可能算ACMer

快速幂（**Exponentiation by squaring**，平方求幂）是一种简单而有效的小算法，它可以以 $O(\log n)$ 的时间复杂度计算乘方。快速幂不仅本身非常常见，而且后续很多算法也都会用到快速幂。

让我们先来思考一个问题：**7的10次方，怎样算比较快？**

方法1： 最朴素的想

这样算无疑太慢了，尤其对计算机的CPU而言，每次运算只乘上一个个位数，无疑太屈才了。这时我们想到，也许可以拆分问题。

方法2：先算7的5次方，即7*7*7*7*7，再算它的平方，共进行了**5次**乘法。

但这并不是最优解，因为对于“7的5次方”，我们仍然可以拆分问题。

方法3：先算7*7得49，则7的5次方为49*49*7，再算它的平方，共进行了**4次**乘法。

模仿这样的过程，我们得到一个在 $O(\log n)$ 时间内计算出幂的算法，也就是快速幂。

递归快速幂

刚刚我们用到的，无非是一个**二分**的思路。我们很自然地可以得到一个递归方程：

$$a^n = \begin{cases} a^{n-1} \cdot a, & \text{if } n \text{ is odd} \\ a^{\frac{n}{2}} \cdot a^{\frac{n}{2}}, & \text{if } n \text{ is even but not 0} \\ 1, & \text{if } n = 0 \end{cases}$$

计算a的n次方，如果n是偶数（不为0），那么就**先计算a的n/2次方，然后平方**；如果n是奇数，那么就**先计算a的n-1次方，再乘上a**；递归出口是**a的0次方为1**。

递归快速幂的思路非常自然，代码也很简单（直接把递归方程翻译成代码即可）：

```
//递归快速幂
int qpow(int a, int n)
{
    if (n == 0)
        return 1;
    else if (n % 2 == 1)
        return qpow(a, n - 1) * a;
    else
    {
        int temp = qpow(a, n / 2);
        return temp * temp;
    }
}
```

注意，这个temp变量是必要的，因为如果不把 $a^{\frac{n}{2}}$ 记录下来，直接写成qpow(a, n /2)*qpow(a, n /2)，那会计算两次 $a^{\frac{n}{2}}$ ，整个算法就退化为了 $O(n)$ 。

在实际问题中，题目常常会要求对一个大素数取模，这是因为计算结果可能会非常巨大，但是在这里考察高精度又没有必要。这时我们的快速幂也应当进行取模，此时应当注意，原则是**步步取模**，如果MOD较大，还应当**开long long**。

```
//递归快速幂（对大素数取模）
#define MOD 1000000007
typedef long long ll;
ll qpow(ll a, ll n)
{
    if (n == 0)
        return 1;
    else if (n %
```

```
        return qpow(a, n - 1) * a % MOD;
    else
    {
        ll temp = qpow(a, n / 2) % MOD;
        return temp * temp % MOD;
    }
}
```

大家知道，递归虽然**简洁**，但会产生**额外的空间开销**。我们可以把递归改写为循环，来避免对栈空间的大量占用，也就是**非递归快速幂**。

非递归快速幂

我们换一个角度来引入非递归的快速幂。还是7的10次方，但这次，我们把10写成**二进制**的形式，也就是 $(1010)_2$ 。

现在我们要计算 $7^{(1010)_2}$ ，可以怎么做？我们很自然地想到可以把它拆分为 $7^{(1000)_2} \cdot 7^{(10)_2}$. 实际上，对于任意的整数，我们都可以把它拆成若干个 $7^{(100...)_2}$ 的形式相乘。而这些 $7^{(100...)_2}$ ，恰好就是 7^1 、 7^2 、 7^4 ……我们只需**不断把底数平方**就可以算出它们。

我们先看代码，再来仔细推敲这个过程：

```
//非递归快速幂
int qpow(int a, int n){
    int ans = 1;
    while(n){
        if(n&1)        //如果n的当前末位为1
            ans
```

```
        a *= a;         //a自乘
        n >>= 1;        //n往右移一位
    }
    return ans;
}
```

最初ans为1，然后我们一位一位算：

1010的最后一位是0，所以a^1这一位不要。然后1010变为101，a变为a^2。

101的最后一位是1，所以a^2这一位是需要的，乘入ans。101变为10，a再自乘。

10的最后一位是0，跳过，右移，自乘。

然后1的最后一位是1，ans再乘上a^8。循环结束，返回结果。

这里的位运算符，>>是右移，表示把二进制数**往右移一位**，相当于/2；&是按位与，&1可以理解为**取出二进制数的最**

的关系了？虽然非递归快速幂因为牵扯到二进制理解起来稍微复杂一点，但基本思路其实和递归快速幂没有太大的出入。

快速幂的拓展

上面所述的都是**整数**的快速幂，但其实，在算 a^n 时，只要a的数据类型支持**乘法**且**满足结合律**，快速幂的算法都是有效的。矩阵、高精度整数，都可以照搬这个思路。下面给出一个模板：

```
//泛型的非递归快速幂
template <typename T>
T qpow(T a, ll n)
{
    T ans = 1; // 赋值为乘法单位元，可能要根据构造函数修改
    while (n)
    {
        if (n & 1)
            ans = ans * a; // 这里就最好别用自乘了，不然重载完*还要重载*=，有点麻烦。
        n >>= 1;
        a = a * a;
    }
    return ans;
}
```

注意，较复杂类型的快速幂的时间复杂度不再是简单的 $O(\log n)$，它与底数的乘法的时间复杂度有关。

例如，**矩阵快速幂**的一个经典应用是求斐波那契数列：

（洛谷P1962） 斐波那契数列

> **题目背景**
> 大家都知道，斐波那契数列是满足如下性质的一个数列：
> $$F_n=\begin{cases}1 & (n\le 2)\\ F_{n-1}+F_{n-2} & (n\ge 3)\end{cases}$$
> **题目描述**
> 请你求出 $F_n \bmod 10^9+7$ 的值。

（以下内容涉及到基本的线性代数知识）

设矩阵 $A=\begin{bmatrix}0 & 1\\ 1 & 1\end{bmatrix}$ ，我们有 $A\begin{bmatrix}F_n\\ F_{n+1}\end{bmatrix}=\begin{bmatrix}F_{n+1}\\ F_n+F_{n+1}\end{bmatrix}=\begin{bmatrix}F_{n+1}\\ F_{n+2}\end{bmatrix}$ ，于是：

$$\begin{aligned}\begin{bmatrix}F_n\\ F_{n+1}\end{bmatrix}&=A\begin{bmatrix}F_{n-1}\\ F_n\end{bmatrix}\\ &=A^2\begin{bmatrix}F_{n-2}\\ F_{n-1}\end{bmatrix}\\ &=\dots\\ &=A^{n-1}\begin{bmatrix}F_1\\ F_2\end{bmatrix}\\ &=A^{n-1}\begin{bmatrix}1\\ 1\end{bmatrix}\end{aligned}$$

这样，我们把原来*

```
#include <cstdio>
#define MOD 1000000007
typedef long long ll;

struct matrix
{
    ll a1, a2, b1, b2;
    matrix(ll a1, ll a2, ll b1, ll b2) : a1(a1), a2(a2), b1(b1), b2(b2) {}
    matrix operator*(const matrix &y)
    {
        matrix ans((a1 * y.a1 + a2 * y.b1) % MOD,
                   (a1 * y.a2 + a2 * y.b2) % MOD,
                   (b1 * y.a1 + b2 * y.b1) % MOD,
                   (b1 * y.a2 + b2 * y.b2) % MOD);
        return ans;
    }
};

matrix qpow(matrix a, ll n)
{
    matrix ans(1, 0, 0, 1); //单位矩阵
    while (n)
    {
        if (n & 1)
            ans = ans * a;
        a = a * a;
        n >>= 1;
    }
    return ans;
}
```

```
int main()
{
    ll x;
    matrix M(0, 1, 1, 1);
    scanf("%lld", &x);
    matrix ans = qpow(M, x - 1);
    printf("%lld\n", (ans.a1 + ans.a2) % MOD);
    return 0;
}
```

编辑于 2020-11-24

算法与数据结构　ACM 竞赛　OI（信息学奥林匹克）

文章被以下专栏收录

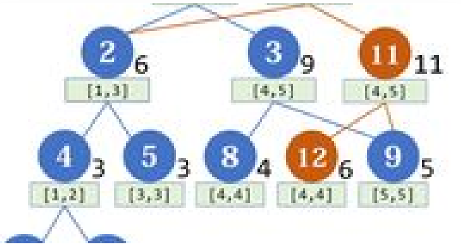
算法学习笔记
记录并分享学习算法的过程

推荐阅读

算法学习笔记(46): 替罪羊树

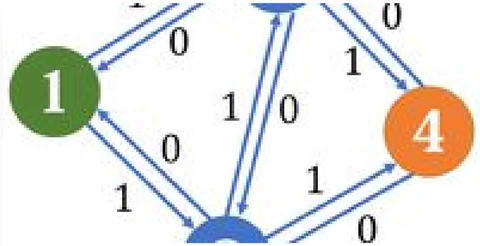
为了防止 二叉搜索树左右不平衡，我们引入平衡树，而其中思路最简单的是替罪羊树（Scapegoat tree）。其保持平衡的方法用一句话就可以概括——当发现某个子树很不平衡时， 暴力重构该子树使...

Pecco　发表于算法学习笔...



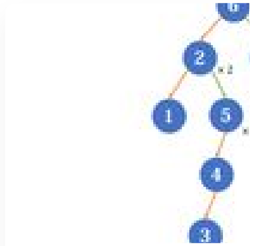
算法学习笔记(50): 可持久化线段树

Pecco　发表于算法学习笔...



算法学习笔记(28): 网络流

Pecco　发表于算法学习笔...



算法学习笔记(4

Pecco　发

35 条评论 ⇄ 切换为时间排序

写下你的评论...

周小迪 2020-09-22

完全看懂了，谢谢作者。😘

4

nlln 2020-10-16

斐波那契数列n≥3那里公式写错了😘

2

Pecco (作者) 回复 nlln 2020-10-16

哈哈居然一直没人发现🤣

1

GreenHand 2020-10-26

作者，我可以转载一部分么，想插进我的题解里面

1

Pecco (作者) 回复 GreenHand 2020-10-26

可以

3

南瓜瓜 10-04

终于看懂了！感谢感谢🤗

赞

▲ 赞同 419 ▼ 35 条评论 分享 喜欢 收藏 申请转载 ···

江城子 07-29

非常感谢，看懂了

❤️

赞

RickDawn 07-20

不用temp,复杂度退化到O(N)怎么算的?

赞

消失的Bliss 回复 RickDawn 08-28

master定理，T(n)=2T(n/2)+O(1)，log{2}{2}=1，O(1)<O(n^(log_{2}{2})),T(n)=O(n^(log_{2}{2}))=O(n)

赞

大嘘 07-05

今天刚看了用矩阵解斐波那契数，没想到接着就刷到快速幂了

赞

六十二 05-30

太强了

赞

做个理智的人 04-14

感动，一名大一电子的 孩子也看懂了!

赞

摘星

有没有大神

赞

優柔不断 回复 摘星 07-11

定义矩阵乘法，看一下类的语法吧

赞

longlong 03-29

我想问下第二种使用离散数学知识的思路是怎么想出来的

赞

何其帅 03-11

矩阵那里的mod有什么用啊☹

赞

风逝 03-06

这种感觉用通项公式去解可能范围更广点（比如系数是变量这样）

赞

思维之火雕玉为灵 2020-12-11

那Python的7**10是什么思路

赞

Pecco (作者) 回复 思维之火雕玉为灵 2020-12-11

没记错的话，Python的**当指数为整数时就会使用快速幂算法

1

云外天际线 2020-11-25

谢谢作者，

赞同 419 35 条评论 分享 喜欢 收藏 申请转载 ···

赞

矢本小季 2020-11-23

非递归快速幂的那个表格写错了，第一行的a 应该是 7^{(1)_2}

赞

Pecco (作者) 回复 矢本小季 2020-11-23

确实，谢谢，稍后改

赞

李二牛 2020-10-28

写的非常棒

赞

1 2 下一页

赞同 419 35 条评论 分享 喜欢 收藏 申请转载 ···