快速幂算法（全网最详细地带你从零开始一步一步优化）_刘扬俊的博客-CSDN博客_快速幂算法

快速幂算法——带你从零开始一步一步优化

目录

什么是快速幂算法

首先，我们先来看一道ACM程序设计题，这道题是杭电OJ中序号为2035的题目，没做过这道题目的同学可以跟着一起做一下（点击此处传送），题目如下：

问题描述：

> 求A^B的最后三位数表示的整数。
> 说明：A^B的含义是“A的B次方”

这道题目乍一看会觉得并不难啊，题目短短一行而已，而且思路也很容易，求幂这种算法一般在初学程序设计语言的时候应该都有联系过，只要写一个简单的循环就能够搞定。

```
1.
2.
3.
4.
5.
6.
7.  long long normalPower(long long base,long long power){
8.  long long result=1;
9.  for(int i=1;i<=power;i++){
10.         result=result*base;
```

```
    }
return result%1000;
}
```

这道题不是分分钟解决吗？接下来，让我们来写一个主函数测试一下：

```
int main(){
long long base,power;
    cin>>base>>power;

    cout<<"base="<<base<<" power="<<power<<" "<<normalPower(base,power)<<endl;

return 0;

}
```

然后，让我们愉快的来求一下2^100的结果的后三位数表示的整数是什么吧！输出结果如下：

```
2 100
base=2 power=100 0
```

为什么答案会是0呢？明明没有错误的啊！~

先不急，我们再来考虑一下，这道题其实出的很有意思，题目要求你输出结果的后三位，为什么不让你直接输出结果呢？难道仅仅只是为了增大题目的难度吗？当然不是，我们在初中就学过“指数爆炸”，下面我们在来回顾一下“指数”的概念：

指数：在乘方a中，其中的a叫做底数，n叫做指数，结果叫幂。

f(x)=a^x ， 随着x单位长度的递增，f(x)会呈“爆炸性”增长。

一张纸对折一次，厚度变成原来的2倍。再对折第二次，变为原来的2的2次方倍即4倍。以此类推，假设纸的厚度为0.1mm，则对折24次以后，长度超过1千米；对折39次达55000千米，超过地球赤道长度；对折42次达44万千米，超过地球至月球的距离；对折51次达22亿千米，超过地球至太阳的距离；对折82次为51113光年，超过银河系半径的长度。

因此，如果题目让你求2的100次方，貌似我们程序设计语言中最大的long lnog类型也无法承载这么大的数值，所以题目才不会要求你输出结果，因为结果可能会非常的大，大到没有任何类型可以承载。所以我们会发现上面的结果为什么是0，因为已经发生溢出了。

那为什么题目要求输出结果的最后三位数表示的整数呢？有的同学可能会问：求一个数的最后三位数表示的整数好办，只要用这个结果进行“取模”运算，让其对1000取模，得到的数就是这个数最后三位数表示的整数。（例如：12345的最后三位数表示的整数是：12345%1000=345）。但是，你这结果都无法求出来，让我怎么进行“取模”运算呢？你这不是瞎闹吗？

别急，我们首先来了解一下“取模”运算的运算法则：（具体的证明感兴趣的同学可以问度娘）

> 1. (a + b) % p = (a % p + b % p) % p （1）
> 2. (a - b) % p = (a % p - b % p) % p （2）
> 3. (a * b) % p = (a % p * b % p) % p （3）

其中我们只要关注第“3”条法则即可：(a * b) % p = (a % p * b % p) % p ，我们仔细研究一下这个运算法则，会发现多个因子连续的乘积取模的结果等于每个因子取模后的乘积再取模的结果。也就是说，我们如果要求：

(a*b*c)%d=(a%d*b%d*c%d)%d;

因此，我们可以借助这个法则，只需要在循环乘积的每一步都提前进行“取模”运算，而不是等到最后直接对结果“取模”，也能达到同样的效果。

所以，我们的代码可以变成这个样子：

```
1.
2.
3.
4.
5.
6.
7.  long long normalPower(long long base,long long power){
8.  long long result=1;
9.  for(int i=1;i<=power;i++){
10.         result=result*base;
11.         result=result%1000;
12.     }
13. return result%1000;
14. }
```

我们再来测试一下，这样又能不能输出结果呢？我们仍然来求一下2^100的后三位是什么：

```
2 100
base=2 power=100 376
```

这一次完美的得到了我们想要的结果。2^100的幂结果的后三位整数位376。

为了打消一些同学对这个运算法则的怀疑，我们再用一个结果比较小的式子来验证一下：我们知道2^10为1024,按理来说，最后输出的结果的后三位数表示的整数应该是24，那么是不是这样呢？我们来试一试：

```
2 10
base=2 power=10 24
```

最后的结果果然是24，所以这个法则是没有问题的。我们把下面的代码提交给OJ看一下是否能通过：

```
#include <iostream>
#include <cmath>

using namespace std;









long long normalPower(long long base, long long power) {
long long result = 1;
for (int i = 1; i <= power; i++) {
        result = result * base;
        result = result % 1000;
    }
return result % 1000;
}

int main() {
long long base, power;
while (true) {
        cin >> base >> power;
if (base == 0 && power == 0) break;
        cout << normalPower(base, power) << endl;
    }
return 0;

}
```

最后的结果是成功Accept了。

再次思考

虽然这个求幂的方法很有用，并且提交给OJ也直接Accept了，但是我们来考虑一下这个算法的时间复杂度，假设我们求2的100次方，那么将会执行100次循环。如果我们分析一下这个算法，就会发现这个算法的时间复杂度为O(N),其中N为指数。求一下小的结果还好，

那如果我们要求2的1000000000次方呢？这个程序可能会运行很久很久，具体会多久呢，让我们来测试一下，测试代码如下：

```
#include <iostream>
#include <cmath>
#include <time.h>

using namespace std;









long long normalPower(long long base, long long power) {
long long result = 1;
for (int i = 1; i <= power; i++) {
        result = result * base;
        result = result % 1000;
    }
return result % 1000;
}

int main() {
clock_t start, finish;

long long base, power;
    cin >> base >> power;
    start = clock();

    cout << normalPower(base, power) << endl;
    finish = clock();

    cout << "the time cost is" << double(finish - start) / CLOCKS_PER_SEC;

return 0;

}
```

结果如图所示：

```
2 1000000000
376
the time cost is17.61
Process finished with exit code 0
```

我们发现，虽然结果是成功求出来了，但是用了将近18秒的时间才求出最后的答案。这效率当然是非常的低下的，更谈不上实际的生产应用了。那么有没有什么好的办法能够对其进行优化呢？接下来就是我们本次的主题了：快速幂算法。

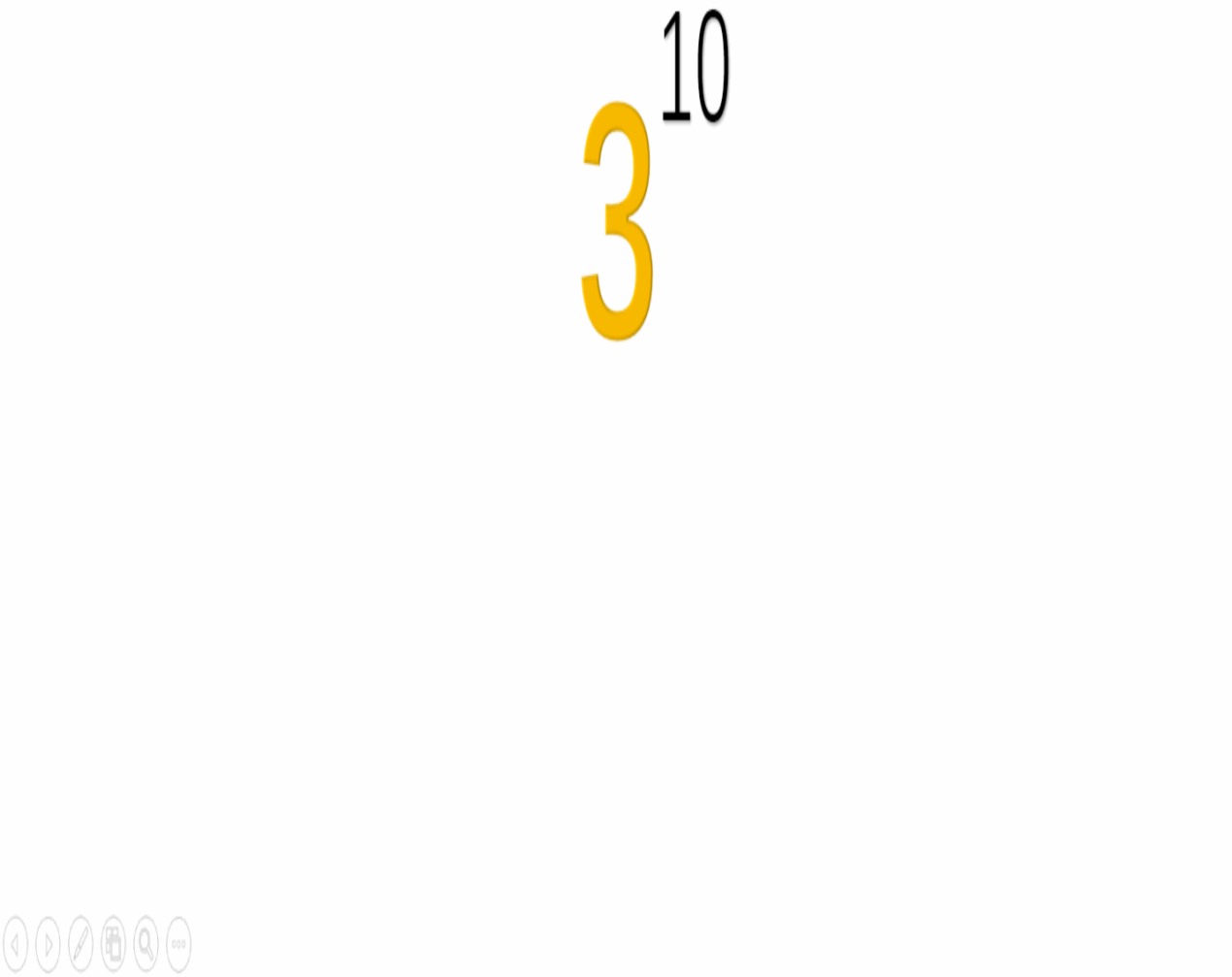
快速幂算法初步入门

快速幂算法能帮我们算出指数非常大的幂，传统的求幂算法之所以时间复杂度非常高（为O(指数n)），就是因为当指数n非常大的时候，需要执行的循环操作次数也非常大。所以我们快速幂算法的核心思想就是每一步都把指数分成两半，而相应的底数做平方运算。这样不仅能把非常大的指数给不断变小，所需要执行的循环次数也变小，而最后表示的结果却一直不会变。让我们先来看一个简单的例子：

> 3^10=3*3*3*3*3*3*3*3*3*3
>
> //尽量想办法把指数变小来，这里的指数为10
>
> 3^10=(3*3)*(3*3)*(3*3)*(3*3)*(3*3)
>
> 3^10=(3*3)^5
>
> 3^10=9^5
>
> //此时指数由10缩减一半变成了5，而底数变成了原来的平方，求3^10原本需要执行10次循环操作，求9^5却只需要执行5次循环操作，但是3^10却等于9^5,我们用一次（底数做平方操作）的操作减少了原本一半的循环量，特别是在幂特别大的时候效果非常好，例如2^10000=4^5000,底数只是做了一个小小的平方操作，而指数就从10000变成了5000，减少了5000次的循环操作。
>
> //现在我们的问题是如何把指数5变成原来的一半，5是一个奇数，5的一半是2.5，但是我们知道，指数不能为小数，因此我们不能这么简单粗暴的直接执行5/2，然而，这里还有另一种方法能表示9^5
>
> 9^5=（9^4）*（9^1）
>
> //此时我们抽出了一个底数的一次方，这里即为9^1，这个9^1我们先单独移出来,剩下的9^4又能够在执行“缩指数”操作了，把指数缩小一半，底数执行平方操作
>
> 9^5=（81^2）*(9^1)
>
> //把指数缩小一半，底数执行平方操作
>
> 9^5=（6561^1）*(9^1)

> //此时，我们发现指数又变成了一个奇数1，按照上面对指数为奇数的操作方法，应该抽出了一个底数的一次方，这里即为6561^1，这个6561^1我们先单独移出来，但是此时指数却变成了0，也就意味着我们无法再进行“缩指数”操作了。
>
> 9^5=（6561^0）*(9^1)*(6561^1)=1*(9^1)*(6561^1)=(9^1)*(6561^1)=9*6561=59049
>
> 我们能够发现，最后的结果是9*6561，而9是怎么产生的？是不是当指数为奇数5时，此时底数为9。那6561又是怎么产生的呢？是不是当指数为奇数1时，此时的底数为6561。所以我们能发现一个规律：最后求出的幂结果实际上就是在变化过程中所有当指数为奇数时底数的乘积。

让我们来看一段简单的动画演示（点击放大）：



接下来，再让我们用代码来演示一下上面的算法：

```
long long fastPower(long long base, long long power) {
long long result = 1;
while (power > 0) {
if (power % 2 == 0) {

            power = power / 2;
            base = base * base % 1000;
        } else {

            power = power - 1;
            result = result * base % 1000;
            power = power / 2;
            base = base * base % 1000;
        }
    }
return result;
}
```

我们再来测试一下此时的快速幂算法和普通的求幂算法的效率，我们仍然来求2的1000000000次方，看一看用时又会是多少：

```
2 1000000000
376
the time cost is0.002
Process finished with exit code 0
```

真让人简直不可思议，竟然只花了0.002秒就求出了结果，而且结果也是376，然而普通的算法却用了将近18秒的时间才求出最后的结果。

压榨性能再优化

虽然上面的快速幂算法效率已经很高了，但是我们仍然能够再一次的对其进行“压榨级别”的优化。我们上面的代码看起来仍然有些地方可以再进一步地进行简化，例如在if和else代码块中仍然有重复性的代码：

```
            power = power / 2;
            base = base * base % 1000;
```

而

```
            power = power - 1;
```

```
2.              power = power / 2;
```

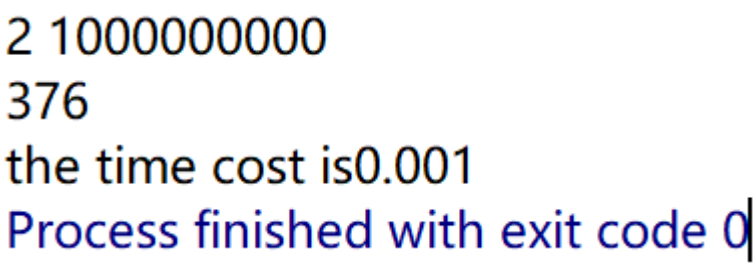
可以压缩成一句：

```
            power = power / 2;
```

因为power是一个整数，例如当power是奇数5时,power-1=4,power/2=2；而如果我们直接用power/2=5/2=2。在整型运算中得到的结果是一样的，因此，我们的代码可以压缩成下面这样：

```
1. long long fastPower(long long base, long long power) {
2. long long result = 1;
3. while (power > 0) {
4. if (power % 2 == 1) {
5.              result = result * base % 1000;
6.          }
7.          power = power / 2;
8.          base = (base * base) % 1000;
9.      }
10. return result;
11. }
```

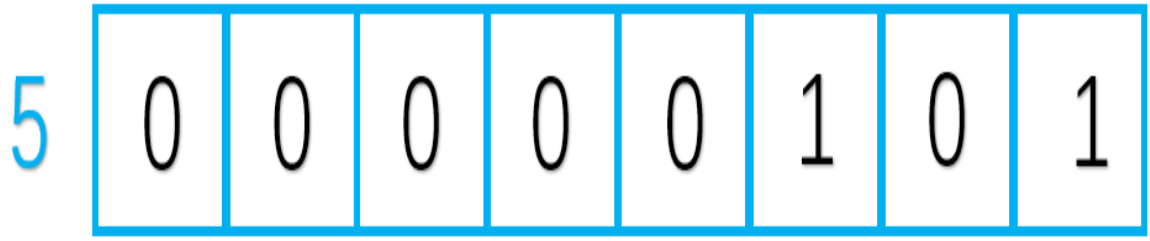
接下来，我们来测试一下优化后的性能如何，仍然是求2的1000000000次方：

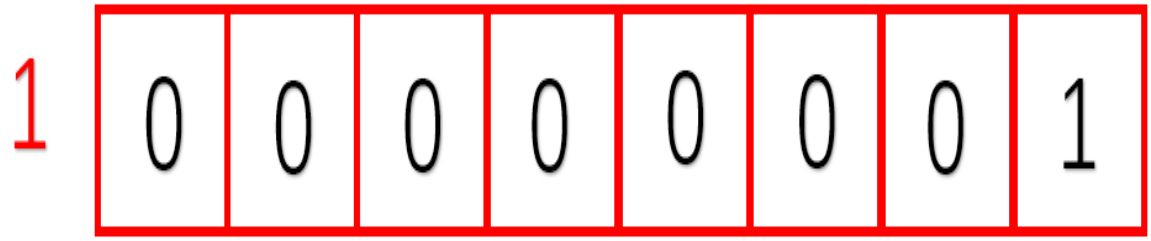


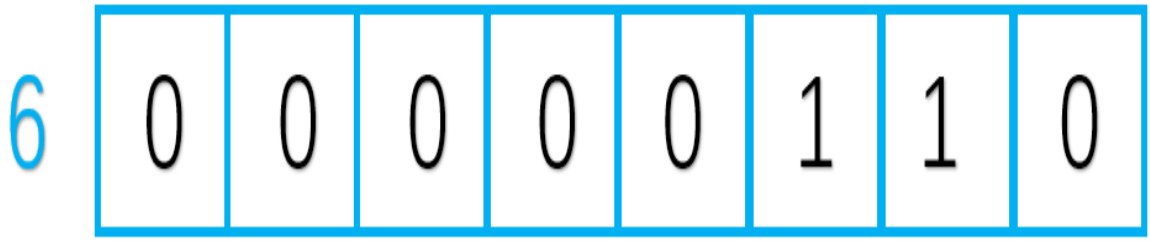
结果仍然是正确的376，但时间上的花费从0.002减少成了0.001。

终极优化

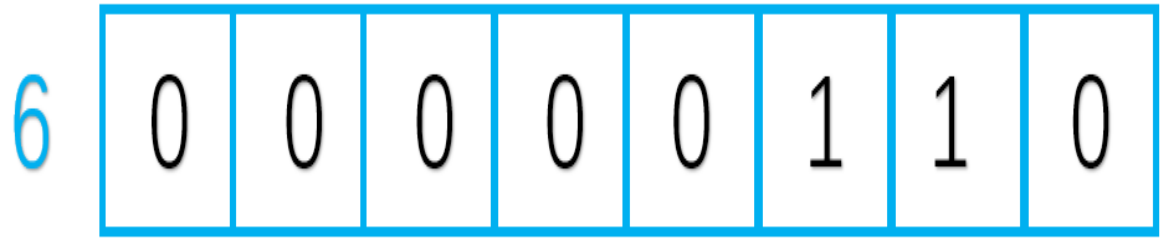
在C语言中，power%2==1可以用更快的“位运算”来代替，例如：power&1。因为如果power为偶数，则其二进制表示的最后一位一定是0；如果power是奇数，则其二进制表示的最后一位一定是1。将他们分别与1的二进制做“与”运算，得到的就是power二进制最后一位的数字了，是0则为偶数，是1则为奇数。例如5是奇数，则5&1=1；而6是偶数，则6&1=0；因此奇偶数的判断就可以用“位运算”来替换了。

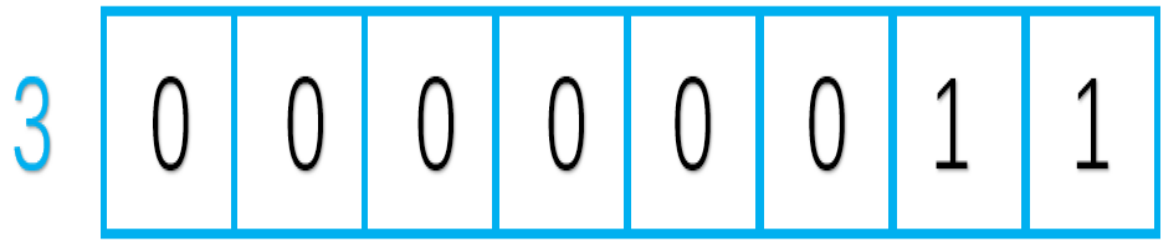






同样，对于power=power/2来说，也可以用更快的“位运算”进行替代，我们只要把power的二进制表示向右移动1位就能变成原来的一半了。





最后，我们的代码就能优化成下面这样：

```
1. long long fastPower(long long base, long long power) {
2. long long result = 1;
3. while (power > 0) {
4. if (power & 1) {
```

```
            result = result * base % 1000;
        }
        power >>= 1;
        base = (base * base) % 1000;
    }
return result;
}
```

我们仍然测试一下求2的1000000000次方，看看终极优化后的代码的性能是怎样的：

```
2 1000000000
376
the time cost is0
Process finished with exit code 0
```

简直可怕，时间花费竟然接近于0秒，我们从最开始的18秒最后压缩到接近0秒，真的是感慨算法的威力！如果同样两家公司，采用不同的算法，给用户带来的体验区别是非常大的，这无不让我们感受到算法的威力。
