<http://www.spoj.com/problems/PON/>

首先，让我们来看看跟素数有关的[费马定理](http://en.wikipedia.org/wiki/Fermat%27s_little_theorem)：

如果 p 是素数，则对于所有整数 a，ap ≡ a (mod p)。

根据上述的费马定理，每当 p 是素数以及 a 不是 p 的倍数时，我们有 ap- 1 ≡ 1 (mod p) 。而且有有效的方法计算 an- 1 mod n ，且只需要 O(log n) 个模 n 乘法运算。因此我们可以确定，当这个关系不成立时，n 不是素数。对于一个给定的数的非素性来说，费马定理是一个强有力的检验。当 n 不是素数时，总是有可能来求 a < n 的一个值，使得 an- 1 ≠ 1 (mod n) 。事实上，经验证明，这样一个值几乎总能非常快地求出。有某些稀少的 n 值，它们经常使得 a n - 1 ≡ 1 (mod n) ，但在此情况下，n 有小于 n1/3 的因子。

我没有找到确定一个很大的数是否素数的有效的算法。但是 [Miller-Rabin primatlity test](http://en.wikipedia.org/wiki/Miller-Rabin_primality_test) 算法能够以很高的概率来检验一个很大的数是否素数。该算法描述如下：

**Input**: n > 3, an odd integer to be tested for primality;   
**Input**: k, a parameter that determines the accuracy of the test  
**Output**: composite if n is composite, otherwise probably prime  
**01:** write n − 1 as 2s·d with d odd by factoring powers of 2 from n − 1  
**02:** LOOP: repeat k times:  
**03:**   pick a randomly in the range [2, n − 2]  
**04:**   x ← ad mod n  
**05:**   if x = 1 or x = n − 1 then do next LOOP  
**06:**   for r = 1 .. s − 1  
**07:**     x ← x2 mod n  
**08:**     if x = 1 then return composite  
**09:**     if x = n − 1 then do next LOOP  
**10:**   return composite  
**11:** return probably prime

构成该算法的思想是，如果 a d ≠ 1 (mod n) 以及 n = 1 + 2s· d 是素数，则值序列

a d mod n，a 2d mod n，a 4d mod n，…，a 2s d mod n

将以 1 结束，而且在头一个 1 的前边的值将是 n – 1 (当 p 是素数时，对于 y 2 ≡ 1 (mod p) ，仅有的解是 y ≡ ±1 (mod p)，因为 (y + 1)(y - 1)必须是 p 的倍数)。注意，如果在该序列中出现了 n – 1，则该序列中的下一个值一定是 1。因为：(n – 1)2  ≡  n2 – 2n + 1  ≡  1  (mod n)。在该算法中：

* 该算法用于判断一个大于 3 的奇数 n 是否素数。参数 k 用于决定 n 是素数的概率。
* 该算法能够肯定地判断 n 是合数，但是只能说 n **可能**是素数。
* 第 01 行，将 n – 1 分解为 2s·d  的形式，这里 d 是奇数。
* 第 02 行，将以下步骤(第 03 到 10 行)循环 k 次。
* 第 03 行，◇在 [2, n - 2] 的范围中独立和随机地选择一个正整数 a 。
* 第 04 行，◇计算该序列的第一个值：x ← ad mod n 。
* 第 05 行，◇如果该序列的第一个数是 1 或者 n - 1，符合上述条件，n 可能是素数，转到第 03 行进行一下次循环。
* 第 06 行，◇循环执行第 07 到 09 行，顺序遍历该序列剩下的 s – 1 个值。
* 第 07 行，◇◇计算该序列的下一个值：x ← x2 mod n 。
* 第 08 行，◇◇如果这个值是 1 ，但是前边的值不是 n - 1，不符合上述条件，因此 n 肯定是合数，算法结束。
* 第 09 行，◇◇如果这个值是 n - 1，因此下一个值一定是 1，符合上述条件，n 可能是素数，转到第 03 行进行下一次循环。
* 第 10 行，◇发现该序列不是以 1 结束，不符合上述条件，因此 n 肯定是合数，算法结束。
* 第 11 行，已经对 k 个独立和随机地选择的 a 值进行了检验，因此判断 n 非常有**可能**是素数，算法结束。

在一次检验中，该算法出错的可能顶多是四分之一。如果我们独立地和随机地选择 a 进行重复检验，一旦此算法报告 n 是合数，我们就可以确信 n 肯定不是素数。但如果此算法重复检验 25 次报告都报告说 n 可能是素数，则我们可以说 n “几乎肯定是素数”。因为这样一个 25 次的检验过程给出关于它的输入的错误信息的概率小于 (1/4)25。这种机会小于 1015 分之一。即使我们以这样一个过程验证了十亿个不同的素数，预料出错的概率仍将小于百万分之一。因此如果真出了错，与其说此算法重复地猜测错，倒不如说由于硬件的失灵或宇宙射线的原因，我们的计算机在它的计算中丢了一位。这样的概率性算法使我们对传统的可靠性标准提出一个问号：我们是否真正需要有素性的严格证明。(以上文字引用自[Donald E. Knuth](http://www-cs-faculty.stanford.edu/~knuth/) 所著的《计算机程序设计艺术 第2卷 半数值算法(第3版)》第 359 页“4.5.4 分解素因子”中的“算法P(概率素性检验)”后面的说明)

#include<cstdio>

#define ll long long

const int Prime[6] = {0, 2, 3, 5, 7, 11};

ll mulmod(ll x, ll y, ll p)

{

ll res = 0;

while (y)

{

if (y & 1) res = (res + x) % p;

x = (x + x) % p;

y >>= 1;

}

return res;

}

ll expmod(ll x, ll y, ll p)

{

ll res = 1;

while (y)

{

if (y & 1) res = mulmod(res, x, p);

x = mulmod(x, x, p);

y >>= 1;

}

return res;

}

bool witness(ll a, ll p)

{

ll y = p - 1;

int cnt = 0;

while (!(y & 1))

{

cnt++;

y >>= 1;

}

ll \_a = expmod(a, y, p);

if (\_a == 1 || \_a == p - 1) return 0;

while (cnt--)

{

ll tmp = mulmod(\_a, \_a, p);

if (tmp == 1 && \_a != p - 1) return 1;

if (tmp == p - 1) return 0;

\_a = tmp;

}

if (\_a != 1) return 1;

return 0;

}

bool isprime(ll x)

{

for (int i = 1; i <= 5; i++)

{

if (x == Prime[i]) return 1;

if (witness(Prime[i], x)) return 0;

}

return 1;

}

int main()

{

int k;

scanf("%d", &k);

while (k--)

{

ll x;

scanf("%lld", &x);

if (isprime(x)) printf("YES\n");

else printf("NO\n");

}

return 0;

}