**表为平方和问题**

简单来说就是对于丢番图方程来说，求满足条件的整数解。当然，根据的范围不同，所使用的方法也可能会不一样，接下来将会以几个题为例进行深度剖析。

**题意：**给定一个圆的方程，其中有，求在这个圆上有多少个整数点。

**分析：**这个问题比较简单，因为对于丢番图方程来说，整数解的个数可以通过如下方法计算



     其中，满足

**（1）**当为偶数时，有

**（2）**当为奇数时，有

     那么，本题的方法就很明确了，先对进行大数分解，然后搜出所有因子，再枚举所有因子判断累加即可。由于比较简单，省略代码。

**题意：**给定一个正整数，其中，求该正整数最少能够使用多少个正整数的平方和来表示。

**分析：**先来认识几个重要的定理，如下

**（1）费马平方和定理**

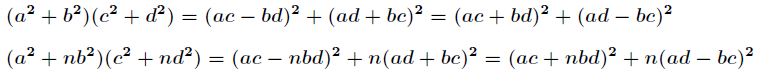
       奇质数能表示为两个平方数之和的充分必要条件是该素数被**4**除余**1**

**（2）费马平方和定理的拓展定理**

       正整数能表示为两平方数之和的充要条件是在它的标准分解式中，形如素因子的指数是偶数

**（3）Brahmagupta–Fibonacci identity**

如果两个整数都能表示为两个平方数之和，则它们的积也能表示为两个平方数之和。公式及拓展公式为



        从这个定理可以看出：如果不能表示为三个数的平方和，那么也就不能表示为两个数的平方和。

**（4）四平方和定理**

每个正整数都可以表示成四个整数的平方数之和

**（5）表为3个数的平方和条件**

正整数能表示为三个数的平方和的充要条件是不能表示成的形式，其中和为非负整数。

      通过上面的几个重要的定理，就可以来做这题了。

1. **int** Work(LL n)
2. {
3. **while**(n % 4 == 0) n >>= 2;
4. **if**(n % 8 == 7) **return** 4;
5. LL i = 8, t = 9;
6. **while**(t <= n)
7. {
8. **while**(n % t == 0) n /= t;
9. i += 8;
10. t += i;
11. }
12. **if**(n == 1) **return** 1;
13. **if**(n % 2 == 0) n >>= 1;
14. **if**(n % 4 == 3) **return** 3;
15. LL k = 3;
16. **while**(k \* k <= n)
17. {
18. **if**(n % k == 0) **return** 3;
19. k += 4;
20. }
21. **return** 2;
22. }
24. **int** main()
25. {
26. LL n;
27. **while**(cin>>n)
28. cout<<Work(n)<<endl;
29. **return** 0;
30. }

上面第**13**行至第**19**行的作用是消去中所有素数的偶次幂，这里为，每次增加**2**，而每次相邻两项满

足，所以每次增加**8**。

 接下来，开始进入我们今天的重点，求丢番图方程的解。这里很大，范围是。

**分析：**通过**Brahmagupta–Fibonacci identity**知道，如果两个整数都能表示为两个平方数之和，则它们的积也

     能表示为两个平方数之和，即



      那么，我们可以先对素因子分解，采用**Pollard-rho**分解法，然后对于每个素数，求出的

     解，然后进行合并即可。而求解丢番图方程（其中为素数）的方法如下

**（1）**首先判断奇素数是否满足条件，如果是才有解，并且解唯一，否则无解。

**（2）**找出二次同余方程的一个解

**（3）**对奇素数和进行欧几里德辗转相除运算，记录每次的余数为（注意第一次的余数为），当满

         足时结束，此时的就是丢番图方程中的，进而通过得到。

1. **const** **int** Times = 10;
2. **const** **int** N = 65;
4. **using** **namespace** std;
5. **typedef** **long** **long** LL;
7. LL ct, cnt;
8. LL fac[N], num[N];
10. **struct** T
11. {
12. LL p, d;
13. };
14. LL w;
16. **struct** Pair
17. {
18. LL x, y;
19. **bool** operator < (**const** Pair &a) **const**
20. {
21. **if**(a.x == x) **return** a.y > y;
22. **return** a.x > x;
23. }
24. };
25. Pair node[N];
26. set<Pair> Set;
27. set<Pair> Mem;
29. **int** sign[2][2] = {{-1, 1}, {1, -1}};
31. LL gcd(LL a, LL b)
32. {
33. **return** b? gcd(b, a % b) : a;
34. }
36. LL multi(LL a, LL b, LL m)
37. {
38. LL ans = 0;
39. a %= m;
40. **while**(b)
41. {
42. **if**(b & 1)
43. {
44. ans = (ans + a) % m;
45. b--;
46. }
47. b >>= 1;
48. a = (a + a) % m;
49. }
50. **return** ans;
51. }
53. LL quick\_mod(LL a, LL b, LL m)
54. {
55. LL ans = 1;
56. a %= m;
57. **while**(b)
58. {
59. **if**(b & 1)
60. {
61. ans = multi(ans, a, m);
62. b--;
63. }
64. b >>= 1;
65. a = multi(a, a, m);
66. }
67. **return** ans;
68. }
70. **bool** Miller\_Rabin(LL n)
71. {
72. **if**(n == 2) **return** **true**;
73. **if**(n < 2 || !(n & 1)) **return** **false**;
74. LL m = n - 1;
75. **int** k = 0;
76. **while**((m & 1) == 0)
77. {
78. k++;
79. m >>= 1;
80. }
81. **for**(**int** i=0; i<Times; i++)
82. {
83. LL a = rand() % (n - 1) + 1;
84. LL x = quick\_mod(a, m, n);
85. LL y = 0;
86. **for**(**int** j=0; j<k; j++)
87. {
88. y = multi(x, x, n);
89. **if**(y == 1 && x != 1 && x != n - 1) **return** **false**;
90. x = y;
91. }
92. **if**(y != 1) **return** **false**;
93. }
94. **return** **true**;
95. }
97. LL pollard\_rho(LL n, LL c)
98. {
99. LL i = 1, k = 2;
100. LL x = rand() % (n - 1) + 1;
101. LL y = x;
102. **while**(**true**)
103. {
104. i++;
105. x = (multi(x, x, n) + c) % n;
106. LL d = gcd((y - x + n) % n, n);
107. **if**(1 < d && d < n) **return** d;
108. **if**(y == x) **return** n;
109. **if**(i == k)
110. {
111. y = x;
112. k <<= 1;
113. }
114. }
115. }
117. **void** find(LL n, **int** c)
118. {
119. **if**(n == 1) **return**;
120. **if**(Miller\_Rabin(n))
121. {
122. fac[ct++] = n;
123. **return** ;
124. }
125. LL p = n;
126. LL k = c;
127. **while**(p >= n) p = pollard\_rho(p, c--);
128. find(p, k);
129. find(n / p, k);
130. }
132. **void** Partion(LL n)
133. {
134. ct = 0;
135. find(n, 120);
136. sort(fac, fac + ct);
137. num[0] = 1;
138. **int** k = 1;
139. **for**(**int** i=1; i<ct; i++)
140. {
141. **if**(fac[i] == fac[i-1])
142. ++num[k-1];
143. **else**
144. {
145. num[k] = 1;
146. fac[k++] = fac[i];
147. }
148. }
149. cnt = k;
150. }
152. **bool** Filter()
153. {
154. **for**(**int** i = 0; i < cnt; i++)
155. {
156. **if**(fac[i] % 4 == 3 && (num[i] & 1))
157. **return** **true**;
158. }
159. **return** **false**;
160. }
162. T multi\_er(T a, T b, LL m)
163. {
164. T ans;
165. ans.p = (multi(a.p, b.p, m) + multi(multi(a.d, b.d, m), w, m)) % m;
166. ans.d = (multi(a.p, b.d, m) + multi(a.d, b.p, m)) % m;
167. **return** ans;
168. }
170. T power(T a, LL b, LL m)
171. {
172. T ans;
173. ans.p = 1;
174. ans.d = 0;
175. **while**(b)
176. {
177. **if**(b & 1)
178. {
179. ans = multi\_er(ans, a, m);
180. b--;
181. }
182. b >>= 1;
183. a = multi\_er(a, a, m);
184. }
185. **return** ans;
186. }
188. LL \_power(LL a, LL b)
189. {
190. LL ans = 1;
191. **while**(b)
192. {
193. **if**(b & 1)
194. {
195. ans = ans \* a;
196. b--;
197. }
198. b >>= 1;
199. a = a \* a;
200. }
201. **return** ans;
202. }
204. LL Legendre(LL a, LL p)
205. {
206. **return** quick\_mod(a, (p - 1) >> 1, p);
207. }
209. LL Work(LL n, LL p)
210. {
211. LL a = -1, t;
212. **while**(1)
213. {
214. a = rand() % p;
215. t = a \* a - n;
216. w = (t % p + p) % p;
217. **if**(Legendre(w, p) + 1 == p) **break**;
218. }
219. T tmp;
220. tmp.p = a;
221. tmp.d = 1;
222. T ans = power(tmp, (p + 1) >> 1, p);
223. **return** ans.p;
224. }
226. **void** Solve(LL n, LL p, LL &\_x, LL &\_y)
227. {
228. LL x = Work(n, p);
229. **if**(x >= p - x)
230. x = p - x;
231. LL t = p;
232. LL r = x;
233. LL limit = (LL)sqrt(1.0 \* p);
234. **while**(r > limit)
235. {
236. x = r;
237. r = t % x;
238. t = x;
239. }
240. \_x = r;
241. \_y = (LL)sqrt((p - r \* r));
242. **if**(\_x > \_y) swap(\_x, \_y);
243. }
245. **void** getData(Pair node[], **int** &\_cnt)
246. {
247. **for**(**int** i = 0; i < cnt; i++)
248. {
249. **if**(fac[i] == 2)
250. {
251. LL res = \_power(fac[i], num[i]>>1);
252. **if**(num[i] & 1)
253. node[\_cnt].x = res;
254. **else**
255. node[\_cnt].x = 0;
256. node[\_cnt].y = res;
257. \_cnt++;
258. **continue**;
259. }
260. **if**(fac[i] % 4 == 3)
261. {
262. LL res = \_power(fac[i], num[i]>>1);
263. node[\_cnt].x = 0;
264. node[\_cnt].y = res;
265. \_cnt++;
266. **continue**;
267. }
268. Solve(-1, fac[i], node[\_cnt].x, node[\_cnt].y);
269. \_cnt++;
270. **for**(**int** j = 1; j < num[i]; j++)
271. {
272. node[\_cnt].x = node[\_cnt - 1].x;
273. node[\_cnt].y = node[\_cnt - 1].y;
274. \_cnt++;
275. }
276. }
277. }
279. **void** dfs(**int** dept, **int** cnt, Pair ans)
280. {
281. **if**(dept == cnt - 1)
282. {
283. Set.insert(ans);
284. **return**;
285. }
286. **for**(**int** i = 0; i < 2; i++)
287. {
288. Pair \_ans;
289. \_ans.x = ans.x \* node[dept + 1].x + sign[i][0] \* ans.y \* node[dept + 1].y;
290. \_ans.y = ans.x \* node[dept + 1].y + sign[i][1] \* ans.y \* node[dept + 1].x;
291. **if**(\_ans.x < 0) \_ans.x = -\_ans.x;
292. **if**(\_ans.y < 0) \_ans.y = -\_ans.y;
293. **if**(\_ans.x > \_ans.y) swap(\_ans.x, \_ans.y);
294. **if**(Mem.find(\_ans) != Mem.end()) **return**;
295. Mem.insert(\_ans);
296. dfs(dept + 1, cnt, \_ans);
297. }
298. }
300. **void** Merge(Pair node[], **int** cnt)
301. {
302. Pair ans;
303. ans.x = node[0].x;
304. ans.y = node[0].y;
305. Mem.insert(ans);
306. dfs(0, cnt, ans);
307. }
309. **int** main()
310. {
311. LL n;
312. **while**(cin>>n)
313. {
314. **if**(n == 1)
315. {
316. puts("0 1");
317. **continue**;
318. }
319. Partion(n);
320. **if**(Filter())
321. {
322. puts("No solution");
323. **continue**;
324. }
325. Set.clear();
326. Mem.clear();
327. **int** \_cnt = 0;
328. getData(node, \_cnt);
329. Merge(node, \_cnt);
330. set<Pair>::iterator it;
331. **for**(it = Set.begin(); it != Set.end(); it++)
332. cout<<it->x<<" "<<it->y<<endl;
333. }
334. **return** 0;

}