**Documentación AC-Knights**

INDEX

[1. Plantilla 3](#_Toc514201625)

[2. DP 4](#_Toc514201626)

[NK 4](#_Toc514201627)

[Longest Common Subsequence 5](#_Toc514201628)

[Longest Common Substring 6](#_Toc514201629)

[3. Geometry 7](#_Toc514201630)

[4. Graph’s Theory 14](#_Toc514201631)

[Dijkstra 14](#_Toc514201632)

[Dinitz 15](#_Toc514201633)

[DSU 17](#_Toc514201634)

[Find Cycles 18](#_Toc514201635)

[Floyd-Warshall 19](#_Toc514201636)

[Kruskal 20](#_Toc514201637)

[Longest Path in DAG 22](#_Toc514201638)

[Maximum Diameter in a Graph 23](#_Toc514201639)

[Maximum Bipartite Matching 24](#_Toc514201640)

[Strongly Connected Components 25](#_Toc514201641)

[Topological Sort 26](#_Toc514201642)

[5. Number Theory 27](#_Toc514201643)

[Fermat Little’s theorem 27](#_Toc514201644)

[Fibonacci 28](#_Toc514201645)

[Prime Numbers 29](#_Toc514201646)

[Triangle of Mahonian 34](#_Toc514201647)

[6. Segment Tree 35](#_Toc514201648)

[Lazy Propagation 35](#_Toc514201649)

[Persistent Segment Tree 36](#_Toc514201650)

[7. Strings 37](#_Toc514201651)

[KMP 37](#_Toc514201652)

[Suffix Array 38](#_Toc514201653)

[Z Function 39](#_Toc514201654)

[8. Variuos 40](#_Toc514201655)

[2-Sat 40](#_Toc514201656)

[Bits 41](#_Toc514201657)

[Fast IO 42](#_Toc514201658)

[FFT 43](#_Toc514201659)

[Fraction 45](#_Toc514201660)

[Hour 47](#_Toc514201661)

[Matrix Exponentiation 49](#_Toc514201662)

[Roman Numbers 50](#_Toc514201663)

[Rotate Matrix 51](#_Toc514201664)

[Sorts 52](#_Toc514201665)

[9. Otros 55](#_Toc514201666)

[Series 55](#_Toc514201667)

[First 5000 digits of PI 56](#_Toc514201668)

[First 150 Fibonacci numbers 57](#_Toc514201669)

[First 1000 prime numbers 59](#_Toc514201670)

[First 25 Catalan numbers 60](#_Toc514201671)

[First 100 powers of 2 61](#_Toc514201672)

[First 30 Rows of Pascal Triangle 62](#_Toc514201673)

[Código ASCII 64](#_Toc514201674)

[Laws and facts 65](#_Toc514201675)

# Plantilla

1. #include <bits/stdc++.h>
2. #include <algorithm>
3. #include <bitset>
4. #include <cmath>
5. #include <cstdio>
6. #include <cstring>
7. #include <deque>
8. #include <fstream>
9. #include <functional>
10. #include <iomanip>
11. #include <iostream>
12. #include <limits.h>
13. #include <map>
14. #include <math.h>
15. #include <numeric>
16. #include <queue>
17. #include <set>
18. #include <sstream>
19. #include <stack>
20. #include <stdio.h>
21. #include <stdlib.h>
22. #include <string>
23. #include <utility>
24. #include <vector>
26. #define PI 3.14159265358979323846
27. #define EPS 1e-6
28. #define INF 1000000000
30. #define \_ ios\_base::sync\_with\_stdio(0), cin.tie(0), cin.tie(0), cout.tie(0), cout.precision(15);
31. #define FOR(i, a, b) for(int i=int(a); i<int(b); i++)
32. #define RFOR(i, a, b) for(int i=int(a)-1; i>=int(b); i--)
33. #define FORC(cont, it) for(decltype((cont).begin()) it = (cont).begin(); it != (cont).end(); it++)
34. #define RFORC(cont, it) for(decltype((cont).rbegin()) it = (cont).rbegin(); it != (cont).rend(); it++)
35. #define pb push\_back
37. using namespace std;
39. typedef long long ll;
40. typedef pair<int, int> ii;
41. typedef vector<int> vi;
43. #define MAXN 10
44. #define MOD 1000000007
46. int main() { \_
48. return 0;
49. }

# DP

## NK

1. ll fact[MAXN], inv[MAXN], pascal[MAXN][MAXN];
3. ll ppow(ll n, ll p) {
4. ll ret = 1;
5. while (p) {
6. if (p & 1) ret = (ret\*n) % MOD;
7. n = (n\*n) % MOD;
8. p>>=1;
9. }
10. return ret;
11. }
13. void generaFact() {
14. fact[0] = 1;
15. FOR(i, 1, MAXN)
16. fact[i] = (fact[i-1] \* i) % MOD;
18. inv[MAXN-1] = ppow(fact[MAXN-1], MOD - 2);
19. for(int i = MAXN-1; i>0; i--)
20. inv[i-1] = inv[i] \* i % MOD;
21. }
23. ll NK(ll n, ll k) {
24. return fact[n] \* inv[k] % MOD \* inv[n-k] % MOD;
25. }
27. // pascal i,j corresponde a la formula de combinatoria i!/(j!\*(i-j)!), AKA. (N, K)
28. void trianguloPascal() {
29. pascal[0][0] = 1;
30. FOR(i, 1, MAXN) {
31. pascal[i][0] = 1;
32. FOR(j, 1, i+1) {
33. pascal[i][j] = (pascal[i - 1][j - 1] + pascal[i - 1][j]) % MOD;
34. }
35. }
36. }
38. int main(){
39. ll n, k;
40. generaFact();
41. trianguloPascal();
42. while (scanf("%lld %lld", &n, &k) != EOF) {
43. printf("%lld**\n**", NK(n, k));
44. printf("%lld**\n**", pascal[n][k]);
45. }
47. return 0;
48. }

## Longest Common Subsequence

1. #define MAXN 1000005
2. #define MOD 1000000007
4. */\* Returns length of LCS for X[0..n-1], Y[0..m-1] \*/*
5. int lcs(string s1, string s2) {
6. int n = s1.length();
7. int m = s2.length();
8. int L[n + 1][m + 1];
9. memset(L, 0, sizeof(L));
11. */\* Following steps build L[n+1][m+1] in bottom up fashion. Note*
12. *that L[i][j] contains length of LCS of X[0..i-1] and Y[0..j-1] \*/*
13. FOR(i, 1, n+1) {
14. FOR(j, 1, m+1) {
15. if (s1[i-1] == s2[j-1])
16. L[i][j] = L[i-1][j-1] + 1;
17. else
18. L[i][j] = max(L[i-1][j], L[i][j-1]);
19. }
20. }
22. */\* L[n][m] contains length of LCS for X[0..n-1] and Y[0..m-1] \*/*
23. return L[n][m];
24. }
26. int main() {
27. string s1, s2;
28. while (cin >> s1 >> s2)
29. cout << lcs(s1, s2) << endl;
30. }

## Longest Common Substring

1. #define MAXN 10
2. #define MOD 1000000007
4. */\* Returns length of longest common substring of X[0..n-1] and Y[0..m-1] \*/*
5. // Create a table to store lengths of longest common suffixes of
6. // substrings.   Notethat LCSuff[i][j] contains length of longest
7. // common suffix of X[0..i-1] and Y[0..j-1]. The first row and
8. // first column entries have no logical meaning, they are used only
9. // for simplicity of program
10. int LCSubStr(string X, string Y) {
11. int n = X.length();
12. int m = Y.length();
14. int lcs[n+1][m+1];
15. memset(lcs, 0, sizeof(lcs));
16. int result = 0;  // To store length of the longest common substring
18. */\* Following steps build lcs[n+1][m+1] in bottom up fashion. \*/*
19. FOR(i, 1, n+1) {
20. FOR(j, 1, m+1) {
21. if (X[i-1] == Y[j-1]) {
22. lcs[i][j] = lcs[i-1][j-1] + 1;
23. result = max(result, lcs[i][j]);
24. }
25. else {
26. lcs[i][j] = 0;
27. }
28. }
29. }
31. return result;
32. }
34. int main() {
35. string X, Y;
36. cin >> X >> Y;
37. cout << LCSubStr(X, Y) << endl;
39. return 0;
40. }

# Geometry

1. typedef long long ll;
3. double degToRad(double theta){
4. return theta \* PI / 180.0;
5. }
7. double radToDeg(double theta){
8. return theta \* 180 / PI;
9. }
11. // \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Struct: Punto \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*
12. struct point {
13. double x, y;
15. point() {}
16. point(double xx, double yy) {
17. x = xx;
18. y = yy;
19. }
20. point inf() {
21. x = INF;
22. y = INF;
23. }
25. double dist(point p) {
26. return hypot(x - p.x, y - p.y);
27. }
29. point rotate(point pivot, point p, double ang) {
30. double s = sin(ang);
31. double c = cos(ang);
33. // translate point back to origin:
34. p.x -= pivot.x;
35. p.y -= pivot.y;
37. // rotate point
38. double xaux = p.x \* c - p.y \* s;
39. double yaux = p.x \* s + p.y \* c;
41. // translate point back:
42. p.x = xaux + pivot.x;
43. p.y = yaux + pivot.y;
44. return p;
45. }
47. double getAngle(point pivot, point p) {
48. return atan2(p.y - pivot.y, p.x - pivot.x);
49. }
51. void swap() {
52. double aux = x;
53. x = y;
54. y = aux;
55. }
57. void swap(point &a, point &b) {
58. point aux = a;
59. a = b;
60. b = aux;
61. }
63. point punto(point const &p) const {
64. return point(x \* p.x, y \* p.y);
65. }
67. double cruz(point const &p) const {
68. return x \* p.y - y \* p.x;
69. }
71. double cruz(point const &p1, point const &p2) const {
72. return (p1.x - x) \* (p2.y - y) - (p1.y - y) \* (p2.x - x);
73. }
75. point operator +(point const &p) const {
76. return point(x + p.x, y + p.y);
77. }
79. point operator -(point const &p) const {
80. return point(x + p.x, y + p.y);
81. }
83. point operator /(double d) const {
84. return point(x / d, y / d);
85. }
87. bool operator <(point const &p) const {
88. return (x - p.x) > EPS && x < p.x ||
89. fabs(x - p.x) < EPS && (y - p.y) > EPS && y < p.y;
90. }
92. bool operator ==(point p) const {
93. return fabs(x - p.x) < EPS && fabs(y - p.y) < EPS;
94. }
96. void print() {
97. cout << x << "**\t**" << y << endl;
98. }
99. };
100. // \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*
102. // \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Struct: line \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*
103. struct line {
104. point p1, p2;
105. double a, b, c;
107. line() {
108. p1.inf();
109. p2.inf();
110. }
111. line(double A, double B, double C) {
112. a = A;
113. b = B;
114. c = C;
116. p1.inf();
117. p2.inf();
118. }
119. line(point pp1, point pp2) {
120. p1 = pp1;
121. p2 = pp2;
123. if (p2 < p1)
124. swap(p1, p2);
126. if(fabs(p1.x - p2.x) < EPS) {
127. a = 1.0;
128. b = 0.0;
129. c = -p1.x;
130. }
131. else {
132. a = -(double)(p1.y - p2.y) / (p1.x - p2.x);
133. b = 1.0;
134. c = -(double)(a \* p1.x) - (b \* p1.y);
135. }
136. }
138. line perpendicular(point p) {
139. double c = -(a \* p.x - b \* p.y);
140. return line(-b, a, c);
141. }
143. bool sonParalelas(line l1, line l2){
144. return fabs(l1.a - l2.a) < EPS &&
145. fabs(l1.b - l2.b) < EPS;
146. }
148. bool sonIguales(line l1, line l2){
149. return sonParalelas(l1, l2) &&
150. fabs(l1.c - l2.c) < EPS;
151. }
153. bool insideLine(point p) {
154. return ((p1.x - EPS <= p.x && p.x <= p2.x + EPS ||
155. p2.x - EPS <= p.x && p.x <= p1.x + EPS)     &&
157. (p1.y - EPS <= p.y && p.y <= p2.y + EPS ||
158. p2.y - EPS <= p.y && p.y <= p1.y + EPS));
159. }
161. bool intercectan(line l1, line l2, point &p){
162. if (sonParalelas(l1, l2))
163. return false;
165. p.x = (l2.b \* l1.c - l1.b \* l2.c) /
166. (l2.a \* l1.b - l1.a \* l2.b);
168. if(fabs(l1.b) > EPS)
169. p.y = -(l1.a \* p.x + l1.c) / l1.b;
170. else
171. p.y = -(l2.a \* p.x + l2.c) / l2.b;
173. return l2.insideLine(p);
174. }
175. };
176. // \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*
178. // \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Struct: triangle \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*
179. struct triangle {
180. point p1, p2, p3;
181. double a, b, c;
182. double A, B, C;
183. double area, perimetro;
184. int type; // equilatero=1, isoseles=2, equilatero=3
185. bool rect; // triangulo rectangulo o no
187. triangle() {}
188. triangle(double aa, double bb, double cc) {
189. a = aa;
190. b = bb;
191. c = cc;
192. sort();
194. act();
195. }
197. triangle(point pp1, point pp2, point pp3) {
198. p1 = pp1;
199. p2 = pp2;
200. p3 = pp3;
202. a = p1.dist(p2);
203. b = p1.dist(p3);
204. c = p2.dist(p3);
205. sort();
207. act();
208. }
210. double innerCircleRadio(){
211. return area / (perimetro \* 2.0);
212. }
214. double outterCircleRadio(){
215. return a \* b \* c / (4 \* area);
216. }
218. int getType() {
219. if(a==b && b==c)        return 1;
220. else if(a==b || b==c)   return 2;
221. return 3;
222. }
224. bool isRight(){
225. return (a\*a + b\*b - c\*c < EPS);
226. }
228. void sort() {
229. if (a > b)
230. swap(a, b);
231. if (a > c)
232. swap(a, c);
233. if (b > c)
234. swap(b, c);
235. }
237. double getPerimetro() {
238. return a + b + c;
239. }
241. double getArea() {
242. double s = perimetro / 2.0;
243. return sqrt(s \* (s-a) \*(s-b) \* (s-c));
244. }
246. double getAngles() {
247. double aa = a \* a;
248. double bb = b \* b;
249. double cc = c \* c;
251. A = acos(bb + cc - aa) / (2 \* b \* c);
252. B = acos(aa + cc - bb) / (2 \* a \* c);
253. C = acos(aa + bb - cc) / (2 \* a \* b);
254. }
256. void act() {
257. type = getType();
258. rect = isRight();
259. perimetro = getPerimetro();
260. area = getArea();
261. getAngles();
262. }
263. };
264. // \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

267. // \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Struct: circle \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*
268. struct circle {
269. point c;
270. double r, circ, area;
272. circle() {
273. c.x = c.y = 0;
274. r=1;
276. act();
277. }
278. circle(point p, double rr){
279. c = p;
280. r = rr;
282. act();
283. }
285. int insideCircle(point p){// 0-Dentro, 1-Borde, 2-Fuera
286. int dx = p.x - c.x, dy = p.y - c.y;
287. int Euc = dx\*dx + dy\*dy, rSq=r\*r;
288. double dist = c.dist(p);
289. if (fabs(dist - r) < EPS)   return 1;
290. if (dist < r)               return 0;
291. return 2;
292. }
294. double getArc(double deg) {
295. return circ \* deg / 360.0;
296. }
297. double getChord(double deg) {
298. return 2.0 \* r \* r \* (1 - cos(degToRad(deg)));
299. }
300. double getSector(double deg) {
301. return area \* deg / 360.0;
302. }
303. double getSegment(double deg) {
304. triangle t(r, r, getChord(deg));
305. return getSector(deg) - t.getArea();
306. }
308. // 0 - No intercectan
309. // 1 - Intersecta 1 punto
310. // 2 - Intersecta 2 puntos
311. // 3 - Mismo circulo
312. // 4 - Circulo dentro del otro
313. int intersectCirc(circle const &cir, point &p1, point &p2) {
314. double d = c.dist(cir.c);
316. if (d > r + cir.r)                      // No se tocan
317. return 0;
318. if ((r - cir.r) < EPS && c == cir.c)    // Mismo circulo
319. return 3;
320. if (d + min(r, cir.r) < max(r, cir.r))  // Circulo contiene otro
321. return 4;
323. double a = (r \* r - cir.r \* cir.r + d \* d) / (2.0 \* d);
324. double h = sqrt(r \* r - a \* a);
326. // find p2
327. point pp1(c.x + (a \* (cir.c.x - c.x)) / d, c.y + (a \* (cir.c.y - c.y)) / d);
329. // find intersection points p3
330. p1 = point(pp1.x + (h \* (cir.c.y - c.y)/ d),
331. pp1.y - (h \* (cir.c.x - c.x)/ d));
333. p2 = point(pp1.x - (h \* (cir.c.y - c.y)/ d),
334. pp1.y + (h \* (cir.c.x - c.x)/ d));
336. if(fabs(d - r - cir.r) < EPS)
337. return 1;
338. return 2;
339. }
341. double getArea() {
342. return PI \* r \* r;
343. }
345. double getCirc() {
346. return PI \* 2. \* r;
347. }
349. bool operator ==(circle const &cir) const{
350. return c == cir.c && fabs(r - cir.r) < EPS;
351. }
353. void act() {
354. area = getArea();
355. circ = getCirc();
356. }
357. };
359. // \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*
361. typedef vector<point> vp;
362. typedef vector<line> vl;
363. typedef vector<triangle> vt;
364. typedef vector<circle> vc;
366. // \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Convex Hull \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*
367. point p0;
369. int orientation(point p, point q, point r) {
370. int val = (q.y - p.y) \* (r.x - q.x) -
371. (q.x - p.x) \* (r.y - q.y);
373. if (val == 0) return 0;  // colinear
374. return (val > 0)? 1: 2; // clock or counterclock wise
375. }
377. bool compAng(const point p1, const point p2) {
378. int o = orientation(p0, p1, p2);
380. return (o == 0 &&
381. hypot(p0.x - p2.x, p0.y - p2.y) >= hypot(p0.x - p1.x, p0.y - p1.y)) ||
382. (o == 2);
383. }
385. bool compY(const point p1, const point p2) {
386. return p1.y < p2.y || (p1.y == p2.y && p1.x < p2.x);
387. }
389. vp convexHull(vp v) {
390. vp c = v;
391. int n = v.size(), mini = 0;
392. p0 = c[0];
394. FOR(i, 1, n) {
395. int y = c[i].y;
397. if (compY(c[i], p0)) {
398. p0 = c[i];
399. mini = i;
400. }
401. }
403. swap(c[0], c[mini]);
405. sort(c.begin() + 1, c.end(), compAng);
407. vp ans;
408. ans.pb(c[0]);
409. ans.pb(c[1]);
410. ans.pb(c[2]);
412. FOR (i, 3, n) {
413. while (orientation(ans[ans.size()-2], ans[ans.size()-1], c[i]) != 2) {
414. ans.erase(ans.end());
415. }
416. ans.pb(c[i]);
417. }
418. ans.pb(ans[0]);
420. return ans;
421. }
422. // \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

425. // \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Convex Hull v2 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*
426. double cross(const point &O, const point &A, const point &B) {
427. return (A.x - O.x) \* (B.y - O.y) - (A.y - O.y) \* (B.x - O.x);
428. }
430. vp convexHull(vp &P) {
431. int n = P.size(), k = 0;
432. vp H(2\*n);
433. sort(P.begin(), P.end());
434. FOR(i, 0, n) {
435. while (k >= 2 && cross(H[k-2], H[k-1], P[i]) <= 0) k--;
436. H[k++] = P[i];
437. }
438. for (int i = n-2, t = k+1; i >= 0; i--) {
439. while (k >= t && cross(H[k-2], H[k-1], P[i]) <= 0) k--;
440. H[k++] = P[i];
441. }
442. H.resize(k);
443. return H;
444. }
445. // \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*


449. // \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Revisa si es CW \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*
450. bool cw(vp v) {
451. double ret = 0;
452. FOR(i, 1, v.size()) {
453. ret += v[i].x \* v[i-1].y - v[i-1].x \* v[i].y;
454. }
455. return ret < 0;
456. }
457. // \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

460. // \*\*\*\*\*\*\*\*\* Revisa si un punto esta dentro en un poligono \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*
461. bool PointInPolygon(vp v, point p) {
462. bool ret = 0;
463. int j = 0;
464. FOR(i, 1, v.size()) {
465. if(((v[i].y > p.y) != (v[j].y > p.y)) &&
466. (p.x < (v[j].x - v[i].x) \* (p.y - v[i].y) / (v[j].y - v[i].y) + v[i].x)
467. )
468. ret = !ret;
469. j++;
470. }
471. return ret;
472. }
473. // \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

476. // \*\*\*\*\*\*\*\*\*\*\*\*\* Calcular perimetro de un poligono \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*
477. double perimetro(vp v) {
478. double ret = 0;
479. FOR(i, 0, v.size() - 1) {
480. ret += v[i].dist(v[i + 1]);
481. }
482. return ret;
483. }
484. // \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

487. // \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Calcular area de un poligono \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*
488. double area(vp v) {
489. double ret = 0;
490. FOR(i, 1, v.size()) {
491. ret += v[i].x \* v[i-1].y - v[i-1].x \* v[i].y;
492. }
493. return fabs(ret) / 2.0;
494. }
495. // \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*



500. int main () { \_
501. point p(1, 2), p2(2,3);
502. cout << p.x << " " << p.y << endl;
504. double ang = p.getAngle(p, p2);
505. cout << p.getAngle(p, p2) \* 180 / PI << endl;
506. p.rotate(p, p2, -ang).print();
507. }

# Graph’s Theory

## Dijkstra

1. typedef long long ll;
2. typedef pair<ll, ll> ii;
3. typedef pair<ii, ll> iii;
4. #define FIRST first.first
5. #define SECOND first.second
6. #define THIRD second
7. typedef vector<ll> vi;
8. typedef vector<ii> vii;
10. #define MAXN 100000
11. #define MOD 1000000007
13. vii edges[MAXN];// first = cost, second = where
14. ii arr[MAXN];// shortest paths to all nodes will be stored here
15. ll n; // size of graph
17. void dijkstra(ll from, ll to) {
18. for(int i = 0; i < n; i++) arr[i].first = INT\_MAX;
19. priority\_queue<iii, vector<iii>, greater<iii> > pq;
20. pq.push(iii(ii(0, -1), from));
21. while(!pq.empty()){
22. iii cur = pq.top();
23. pq.pop();
24. if(arr[cur.THIRD].first == INT\_MAX)
25. arr[cur.THIRD] = cur.first;
26. else
27. continue;
28. for(int i = 0; i < edges[cur.THIRD].size(); i++){
29. ll cost = edges[cur.THIRD][i].first,
30. where = edges[cur.THIRD][i].second;
31. if(arr[where].first == INT\_MAX)
32. pq.push(iii(ii(cur.FIRST + cost, cur.THIRD), where));
33. }
34. }
35. }

## Dinitz

1. #define MAXN 1000
2. #define MOD 1000000007
4. int n, m, src, dest;
5. int dist[MAXN], q[MAXN], work[MAXN];
7. struct Edge {
8. int to, rev;
9. int f = 0, cap;
11. Edge() {}
12. Edge(int t, int r, int c) :
13. to(t), rev(r), cap(c) {}
14. };
15. vector<Edge> graph[MAXN];
17. void addEdge(int s, int t, int cap) {
18. graph[s].pb(Edge(t, graph[t].size(), cap));
19. graph[t].pb(Edge(s, graph[s].size() - 1, 0));
20. }
22. bool dinitz\_bfs() {
23. memset(dist, -1, sizeof(dist));
24. dist[src] = 0;
25. int qt = 0;
26. q[qt++] = src;
27. FOR(qh, 0, qt) {
28. int u = q[qh];
29. FOR(j, 0, graph[u].size()) {
30. Edge &e = graph[u][j];
31. int v = e.to;
32. if (dist[v] < 0 && e.f < e.cap) {
33. dist[v] = dist[u] + 1;
34. q[qt++] = v;
35. }
36. }
37. }
38. return dist[dest] >= 0;
39. }
41. int dinitz\_dfs(int u, int f) {
42. if (u == dest)  return f;
44. for (int &i = work[u]; i < (int) graph[u].size(); i++) {
45. Edge &e = graph[u][i];
46. if (e.cap <= e.f)   continue;
47. int v = e.to;
48. if (dist[v] == dist[u] + 1) {
49. int df = dinitz\_dfs(v, min(f, e.cap - e.f));
50. if (df > 0) {
51. e.f += df;
52. graph[v][e.rev].f -= df;
53. return df;
54. }
55. }
56. }
57. return 0;
58. }
60. int maxFlow() {
61. int result = 0;
62. while (dinitz\_bfs()) {
63. memset(work, 0, sizeof(work));
64. while (int delta = dinitz\_dfs(src, INT\_MAX))
65. result += delta;
66. }
67. return result;
68. }

71. int main() { \_
72. int n, m, a, b, c;
73. cin >> n >> m >> src >> dest;
74. FOR(i, 0, m) {
75. cin >> a >> b >> c;
76. graph[a].pb(Edge(b, graph[b].size(), c));
77. graph[b].pb(Edge(a, graph[a].size(), c));
78. }
79. cout << maxFlow() << endl;
80. return 0;
81. }

## DSU

1. #define MAXN 10
2. #define MOD 1000000007
4. struct DSU {
5. int bel[MAXN];
6. vi s[MAXN];
8. void reset() {
9. FOR(i, 0, MAXN)
10. bel[i] = 0, s[i].clear();
11. }
13. void unir(int a, int b) {
14. if (a > b)  swap(a, b);
16. FOR(i, 0, s[a].size()) {
17. s[b].pb(s[a][i]);
18. bel[a] = b;
19. }
20. s[a].clear();
21. }
23. int belongs(int x) {
24. return bel[x];
25. }
26. };
28. struct DSU2 {
29. int numSets = 0;
30. int setSize[MAXN];
31. int parent[MAXN];
32. int rank[MAXN];
34. UnionFind(int n) {
35. numSets = n;
36. FOR(i, 0, n)    parent[i] = i, setSize[i] = rank[i] = 0;
37. }
39. void make\_set(int i) {
40. parent[i] = i;
41. rank[i] = 0;
42. }
44. int find\_set(int i) {
45. if (i != parent[i])
46. parent[i] = find\_set(parent[i]);
47. return parent[i];
48. }
50. void unionSet(int i, int j) {
51. if (!isSameSet(i, j)) {
52. numSets--;
53. int x = find\_set(i), y = find\_set(j);
55. if (rank[x] > rank[y]) {
56. parent[y] = x;
57. setSize[x] += setSize[y];
58. }
59. else {
60. parent[x] = y;
61. setSize[y] += setSize[x];
63. if (rank[x] == rank[y])     rank[y]++;
64. }
65. }
66. }
68. bool isSameSet(int i, int j) {
69. return find\_set(i) == find\_set(j);
70. }
71. };

## Find Cycles

1. #define maxN 100005
3. vi edges[maxN];
4. bool cur[maxN], visit[maxN];
5. stack<int> ans;
7. int findCycle(int n) {
8. if (cur[n])     return n;
9. if (visit[n])   return -1;
11. cur[n] = true;
12. visit[n] = true;
14. FOR(i, 0, edges[n].size()) {
15. if(ans.size())  break;
17. int v = findCycle(edges[n][i]);
18. if (v != -1) {
19. cur[n] = false;
20. ans.push(n);
21. if(v == n) {
22. return -1;
23. }
24. return v;
25. }
26. }
27. cur[n] = false;
28. return -1;
29. }
31. int main() {
32. int n, m, a, b;
33. while (cin >> n >> m) {
34. memset(cur, false, sizeof(cur));
35. memset(visit, false, sizeof(visit));
37. FOR(i, 0, m) {
38. cin >> a >> b;
39. edges[a].pb(b);
40. }
42. FOR(i, 0, n) {
43. findCycle(i);
44. }
46. if(! ans.empty()) {
47. cout << "YES**\n**";
48. while(! ans.empty()) {
49. int val = ans.top();
50. ans.pop();
51. cout << val << " **\n**"[ans.empty()];
52. }
53. }
54. else {
55. cout << "NO**\n**";
56. }
57. }
58. return 0;
59. }

## Floyd-Warshall

1. #define MAXN 10005
2. #define MOD 1000000007
4. int graph[MAXN][MAXN];
5. int dist[MAXN][MAXN];
7. void floydWarshall(int n) { // O(n^3)
8. FOR(i, 0, n) FOR(j, 0, n) dist[i][j] = graph[i][j];
10. FOR(k, 0, n) {
11. FOR(i, 0, n) {
12. FOR(j, 0, n) {
13. dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j]);
14. }
15. }
16. }
17. }
19. int main() {
20. FOR(i, 0, MAXN) FOR(j, 0, MAXN) dist[i][j] = INF;
22. int n, m;
23. cin >> n >> m;
24. FOR(i, 0, m) {
25. cin >> a >> b >> c;
26. graph[a][b] = c;
27. }
29. floydWarshall(n);
30. return 0;
31. }

## Kruskal

1. #define MAXN 10005
2. #define MAXM 10005
3. #define MOD 1000000007
5. struct Edge {
6. int from, to, w;
7. Edge() {}
8. Edge(int a, int b, int c) : from(a), to(b), w(c) { }
9. bool operator <(Edge const& e) const {
10. return w < e.w;
11. }
12. };
14. struct UnionFind {
15. int numSets = 0;
16. int setSize[MAXN];
17. int parent[MAXN];
18. int rank[MAXN];
20. UnionFind(int n) {
21. numSets = n;
22. FOR(i, 0, n)    parent[i] = i;
23. }
25. void make\_set(int i) {
26. parent[i] = i;
27. rank[i] = 0;
28. }
30. int find\_set(int i) {
31. if (i != parent[i])
32. parent[i] = find\_set(parent[i]);
33. return parent[i];
34. }
36. void unionSet(int i, int j) {
37. if (!isSameSet(i, j)) {
38. numSets--;
39. int x = find\_set(i), y = find\_set(j);
41. if (rank[x] > rank[y]) {
42. parent[y] = x;
43. setSize[x] += setSize[y];
44. }
45. else {
46. parent[x] = y;
47. setSize[y] += setSize[x];
49. if (rank[x] == rank[y])     rank[y]++;
50. }
51. }
52. }
54. bool isSameSet(int i, int j) {
55. return find\_set(i) == find\_set(j);
56. }
57. };
59. vector<Edge> v;
60. vector<Edge> tree;
62. int kruskal(int n, int m) { // O(ELogE + ELogV)
63. sort(v.begin(), v.end());
65. int mst\_w = 0;
66. UnionFind UF(n);
68. FOR(i, 0, m) {
69. Edge e = v[i];
71. if (! UF.isSameSet(e.from, e.to)) {
72. mst\_w += e.w;
73. UF.unionSet(e.from, e.to);
74. tree.pb(Edge(e.from, e.to, e.w));
75. }
76. }
78. return mst\_w;
79. }
81. int main() {
82. int n, m;
83. int a, b, c;
84. cin >> n >> m;
85. FOR(i, 0, m) {
86. cin >> a >> b >> c;
87. v.pb(Edge(a, b, c));
88. }
90. cout << kruskal(n, m) << endl;
92. return 0;
93. }

## Longest Path in DAG

1. #define MAXN 10
2. #define MOD 1000000007
4. vii graph[MAXN];
5. stack<int> s;
6. bool v[MAXN];
7. int dist[MAXN], n, m;
9. void topologicalSortUtil(int act) {
10. v[act] = 1;
12. FOR(i, 0, graph[act].size())
13. if (!v[graph[act][i].second])  topologicalSortUtil(graph[act][i].second);
15. s.push(act);
16. }
18. void longestPath(int source) {
19. memset(v, 0, sizeof(v));
21. FOR(i, 0, n)
22. if (!v[i])  topologicalSortUtil(i);
24. fill(dist, dist + n, -INF);
25. dist[source] = 0;
27. while (! s.empty()) {
28. int u = s.top();
29. s.pop();
31. if (dist[u] != -INF) {
32. FOR(i, 0, graph[u].size()) {
33. dist[graph[u][i].second] = max(dist[graph[u][i].second], dist[u] + graph[u][i].first);
34. }
35. }
36. }
38. FOR(i, 0, n)
39. if (dist[i] == -INF)    cout << "INF" << endl;
40. else                    cout << dist[i] << endl;
41. }

44. int main() { \_
45. int a, b, c;
46. while(cin >> n >> m) {
47. FOR(i, 0, m) {
48. cin >> a >> b >> c;
49. graph[a].pb(ii(c, b));
50. }
52. int source;
53. cin >> source;
54. longestPath(source);
55. }
57. return 0;
58. }

## Maximum Diameter in a Graph

1. #define MAXN 10
2. #define MOD 1000000007
4. // v es visitados localmente.
5. // v2 es todos los que ya he visitado (para grafos no totalmente conectados)
6. bool v[MAXN], v2[MAXN];
7. vi edges[MAXN];
9. ii bfs(int x, bool b) {
10. queue<ii> q;
11. q.push(ii(x, 0));
13. v[x] = !b;
14. v2[x] |= !b;
15. ii last;
16. while(! q.empty()) {
17. last = q.front();
18. q.pop();
20. FOR(i, 0, edges[last.first].size()) {
21. if (v[edges[last.first][i]] == b) {
22. q.push(ii(edges[last.first][i], last.second + 1));
23. v[edges[last.first][i]] = !b;
24. v2[edges[last.first][i]] |= !b;
25. }
26. }
27. }
29. return last;
30. }
32. int maxDiameter(int x) {
33. ii ret = bfs(x, false);
34. return bfs(ret.first, true).second;
35. }

## Maximum Bipartite Matching

1. #define MAXN 10005
2. #define MOD 1000000007
4. int n, m;
5. int matchL[MAXN], matchR[MAXN];
6. vi edge[MAXN];
7. bool v[MAXN];
9. bool dfs(int from) {
10. if (v[from])   return 0;
11. v[from] = 1;
13. FOR(i, 0, edge[from].size()) {
14. int to = edge[from][i];
16. if (matchR[to] == -1 || dfs(matchR[to])) {
17. matchL[from] = to;
18. matchR[to] = from;
19. return 1;
20. }
21. }
22. return 0;
23. }
25. ll MBM() {
26. ll ans = 0;
27. bool b = 1;
29. memset(matchL, -1, sizeof(matchL));
30. memset(matchR, -1, sizeof(matchR));
32. while (b) {
33. b = 0;
34. memset(v, 0, sizeof(v));
35. FOR(i, 0, n) {
36. if (matchL[i] == -1 && dfs(i)) {
37. ans ++;
38. b = 1;
39. }
40. }
41. }
42. return ans;
43. }
45. int main() { \_
46. int a, b;
47. cin >> n >> m;
48. FOR(i, 0, m) {
49. cin >> a >> b;
50. edge[a].pb(b);
51. }
52. cout << MBM() << endl;
54. return 0;
55. }

## Strongly Connected Components

1. #define MAXN 1000
2. #define MOD 1000000007
4. int n, m;
6. vi edge[MAXN];
7. int disc[MAXN], low[MAXN], belongs[MAXN];
8. bool v[MAXN];
9. stack<int> st;
10. int ttime, comp;
12. void SCCUtil(int u) {
13. disc[u] = low[u] = ttime ++;
14. st.push(u);
15. v[u] = true;
17. FOR(i, 0, edge[u].size()) {
18. int to = edge[u][i];
20. if (disc[to] == -1)
21. SCCUtil(to);
23. if (v[to])
24. low[u]  = min(low[u], low[to]);
25. }
27. if (low[u] == disc[u]) {
28. comp ++;
29. while (1) {
30. int t = st.top();
31. st.pop();
32. belongs[t] = comp;
33. v[t] = false;
35. if (t == u) break;
36. }
37. }
38. }
40. void SSC() {
41. ttime = comp = 0;
42. memset(disc, -1, sizeof(disc));
43. memset(low, -1, sizeof(low));
44. memset(v, 0, sizeof(v));
46. FOR(i, 0, n)
47. if (disc[i] == -1)
48. SCCUtil(i);
49. }

## Topological Sort

1. #define MAXN 10005
2. #define MOD 1000000007
4. vi edges[MAXN];
5. stack<int> s;
6. bool v[MAXN];
8. void topologicalSortUtil(int act) {
9. v[act] = 1;
11. FOR(i, 0, edges[act].size())
12. if (!v[edges[act][i]])  topologicalSortUtil(edges[act][i]);
14. s.push(act);
15. }
17. void topologicalSort(int n) {
18. memset(v, 0, sizeof(v));
20. FOR(i, 0, n)
21. if (!v[i])  topologicalSortUtil(i);
23. while (!s.empty()) {
24. int a = s.top();
25. s.pop();
26. cout << a << " **\n**"[s.empty()];
27. }
28. }
30. int main() {
31. int n, m;
32. int a, b;
33. while (cin >> n >> m) {
34. FOR(i, 0, MAXN)     edges[i].clear();
36. FOR(i, 0, m) {
37. cin >> a >> b;
38. edges[a].pb(b);
39. }
41. topologicalSort(n);
42. }
44. return 0;
45. }

# Number Theory

## Fermat Little’s theorem

1. ll mulmod(ll a, ll b, ll c) { // returns (a \* b) % c, and minimize overflow
2. return (ll)((\_\_int128)(a) \* (b) % c);
3. }
5. ll fastPow(ll x, ll n, ll c) { // returns (a \*\* b) % c, and minimize overflow
6. ll ret = 1;
7. while (n) {
8. if (n & 1) ret = mulmod(ret, x, c);
9. x = mulmod(x, x, c);
10. n >>= 1;
11. }
12. return ret;
13. }


17. */\*\* return modular multiplicative of: a mod p, assuming p is prime \*\*/*
18. ll modInverse(ll a, ll p) {
19. return fastPow(a, p - 2, p);
20. }
22. */\*\* return C(n,k) mod p, assuming p is prime \*\*/*
23. ll modBinomial(ll n, ll k) {
24. ll numerator = 1; // n\*(n-1)\* ... \* (n-k+1)
25. FOR(i, 0, k)
26. numerator = (numerator \* (n - i)) % MOD;
28. ll denominator = 1; // k!
29. FOR (i, 1, k+1)
30. denominator = (denominator \* i) % MOD;
32. ll res = modInverse(denominator, MOD);
33. res = (res \* numerator) % MOD;
34. return res;
35. }
37. ll extendedGCD(ll a, ll b, ll &inva, ll &invb) {
38. if (b) {
39. ll g = extendedGCD(b, a%b, invb, inva);
40. invb = invb - a / b\*inva;
41. return g;
42. }
43. inva = 1, invb = 0;
44. return a;
45. }
47. int main() {
48. ll inva=0, invb=0;
49. int c = extendedGCD(62424, 13, inva, invb);
50. cout << modInverse(62424, 13) << endl;
51. cout << (inva + 7) % 7 << " " << (invb + 62424) % 13 << " " << c << endl;
52. return 0;
53. }

## Fibonacci

1. const long double PHI ((1.0 + sqrt(5.0)) / 2.0);
3. ll O1(ll n) {
4. return (ll)(floor(pow(PHI, n) / sqrt(5.0) + 0.5));
5. }
7. void mult(ll F[2][2], ll M[2][2]) {
8. ll x =  (F[0][0] \* M[0][0]) % MOD + (F[0][1] \* M[1][0]) % MOD;
9. ll y =  (F[0][0] \* M[0][1]) % MOD + (F[0][1] \* M[1][1]) % MOD;
10. ll z =  (F[1][0] \* M[0][0]) % MOD + (F[1][1] \* M[1][0]) % MOD;
11. ll w =  (F[1][0] \* M[0][1]) % MOD + (F[1][1] \* M[1][1]) % MOD;
12. F[0][0] = x % MOD;
13. F[0][1] = y % MOD;
14. F[1][0] = z % MOD;
15. F[1][1] = w % MOD;
16. }
18. void power(ll F[2][2], ll n) {
19. if (n == 0 || n == 1) return;
21. ll M[2][2] = {{1, 1}, {1, 0}};
22. power(F, n / 2);
23. mult(F, F);
25. if (n % 2)
26. mult(F, M);
27. }
29. ll OlnMat(ll n) {
30. if (n == 0) return 0;
32. ll F[2][2] = {{1,1},{1,0}};
33. power(F, n - 1);
35. return F[0][0];
36. }
38. */\*\* Suma de los primeros n numeros de fibo \*\*/*
39. ll fiboSuma(ll n) {
40. return (MOD + OlnMat(n + 2) - 1) % MOD;
41. }
43. ll On(ll n) {
44. if (n == 1) return 0;
46. ll a = 0, b = 1, c = 1;
48. FOR(i, 2, n) {
49. c = b + a;
50. a = b;
51. b = c;
52. }
53. return c;
54. }
56. ll fibo(ll n) {
57. if (n < 72)     return O1(n);
58. if (n < 150)    return On(n);
60. return OlnMat(n);
61. }
63. int main(){
64. ll n;
65. cin >> n;
66. cout << fibo(n) << endl;

## Prime Numbers

1. int pc = 0, f[MAXN], primes[MAXN / 10];

4. // \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  IS PRIME()   \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*
5. //Para TODOS los primos
7. ll mulmod2(ll a, ll b, ll c) { // returns (a \* b) % c, and minimize overflow
8. ll x = 0, y = a % c;
9. while (b) {
10. if (b & 1) x = (x + y) % c;
11. y = (y << 1) % c;
12. b >>= 1;
13. }
14. return x % c;
15. }
17. ll mulmod(ll a, ll b, ll c) {
18. return (ll)((\_\_int128)(a) \* b % c);
19. }
21. ll fastPow(ll x, ll n, ll c) { // returns (a \*\* b) % c, and minimize overflow
22. ll ret = 1;
23. while (n) {
24. if (n & 1) ret = mulmod(ret, x, c);
25. x = mulmod(x, x, c);
26. n >>= 1;
27. }
28. return ret;
29. }
31. // Miller-Rabin primality test
32. bool millerRabin(ll n) {
33. ll d = n - 1;
34. int s = 0;
35. while (d % 2 == 0) {
36. s++;
37. d >>= 1;
38. }
39. // It's garanteed that these values will work for any number smaller than 2^64
40. int a[12] = { 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37 };
41. FOR(i, 0, 12)   if(n == a[i])   return true;
43. FOR(i, 0, 12) {
44. bool comp = fastPow(a[i], d, n) != 1;
45. if(comp) FOR(j, 0, s) {
46. ll fp = fastPow(a[i], (1LL << (ll)j)\*d, n);
47. if (fp == n - 1) {
48. comp = false;
49. break;
50. }
51. }
52. if(comp) return false;
53. }
54. return true;
55. }
57. // Miller-Rabin primality test
58. ll rN(){
59. return maxR \* rand() + rand();
60. }
62. bool miller(ll n, ll d){
63. ll a = 2 + rN() % (n - 4);
64. ll x = fastPow(a, d, n);
65. if(x == 1 || x == n - 1)
66. return true;
67. while(d < n - 1){
68. x = \_\_int128(x) \* x % n;
69. if(x == 1)
70. return false;
71. if(x == n - 1)
72. return true;
73. d <<= 1;
74. }
75. return false;
76. }
78. bool isPrime(ll n, int k){
79. if(n == 3 || n == 2)
80. return true;
81. if(n == 1 || n % 2 == 0)
82. return false;
83. ll d = n - 1;
84. while(d % 2 == 0)
85. d >>= 1;
86. while(k--){
87. if(!miller(n, d))
88. return false;
89. }
90. return true;
91. }
92. // \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

95. // \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  GET PRIME FACTORS()   \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*
96. */\*\**
97. *Devuele los factores primos de un numero. No olvidar hacer unique para quitar repetidos*
98. *int tamano = unique(primeFactors.begin(), primeFactors.end()) - primeFactors.begin();*
99. *\*\*/*
101. ll A, B;
102. vl primeFactors;
104. ll funcRand(ll x, ll n) {
105. return (mulmod(x, (x + A), n) + B) % n;
106. }
108. void pollardRho(ll n) {
109. if (n == 1) return;
110. if (millerRabin(n)) {
111. primeFactors.pb(n);
112. return;
113. }
114. ll d = n, x, y;
115. while(d == n) {
116. d = 1;
117. x = y = 2;
118. A = 2 + rand() % (n - 2);
119. B = 2 + rand() % (n - 2);
121. while (d == 1) {
122. x = funcRand(x, n);
123. y = funcRand(funcRand(y, n), n);
124. d = \_\_gcd(abs(x - y), n);
125. }
126. }
128. if (n / d != d)
129. pollardRho(n / d);
130. pollardRho(d);
131. }
132. // \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*
134. // \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  GET SMALLEST DIV PRIME()   \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*
135. // Obtener f[t], el primo mas chico que divide a f[t]
137. int divPrime[MAXN];
139. void getSmallestDivPrime() {
140. FOR(i, 2, MAXN) {
141. if (divPrime[i] == -1) {
142. for (int j = 1; i\*j < MAXN; j++) if (divPrime[i\*j] == -1) divPrime[i\*j] = i;
143. }
144. }
145. }
146. // \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

149. // \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  GET DIVISORS()   \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*
150. vll factorsOfN;
151. vl divisors;
153. // Usar cuando se necesita sacar muchos numeros
154. void getFactorsOfN2(int n) {
155. int t = n;
156. int last = -1;
157. while (divPrime[t] != -1) {
158. if (divPrime[t] == last)    factorsOfN[factorsOfN.size() - 1].second ++;
159. else                        factorsOfN.pb(ii(divPrime[t], 1));
160. last = divPrime[t];
161. t /= divPrime[t];
162. }
163. }
165. // Usar cuando es numeros muy grandes
166. void getFactorsOfN(ll n) {
167. pollardRho(n);
168. sort(primeFactors.begin(), primeFactors.end());
169. int ss = unique(primeFactors.begin(), primeFactors.end()) - primeFactors.begin();
170. int s = 0;
171. ll num = n;
172. FOR(i, 0, ss) {
173. bool b = false;
174. while (num % primeFactors[i] == 0) {
175. if (b)  factorsOfN[s].second ++;
176. else    factorsOfN.pb(pll(primeFactors[i], 1)), b = 1;
178. num /= primeFactors[i];
179. }
180. s ++;
181. }
182. }
184. void getDivisorsOfN(ll n) {
185. primeFactors.clear();
186. factorsOfN.clear();
187. divisors.clear();
189. getFactorsOfN(n);
190. divisors.push\_back(1);
192. FOR(i, 0, factorsOfN.size()) {
193. int s = divisors.size(), num = 1;
195. FOR(j, 0, factorsOfN[i].second) {
196. num \*= factorsOfN[i].first;
197. FOR(k, 0, s) divisors.push\_back(divisors[k] \* num);
198. }
199. }
200. sort(divisors.begin(), divisors.end());
201. FOR(i, 0, divisors.size())  cout << divisors[i] << endl;
202. }
203. // \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

206. // \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Criba de Eratosthenes \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*
207. void Eratosthenes() {
208. bool isPrime[MAXN];
209. memset(isPrime, 0, sizeof(isPrime));
210. FOR(i, 2, MAXN) {
211. if (!isPrime[i]) {
212. primes[pc ++] = i;
213. for (int j = 2; j\*i < MAXN; j++) {
214. isPrime[i\*j] = true;
215. }
216. }
217. }
218. }
219. // \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

222. // \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Criba de Eratosthenes O(N) \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*
223. void EratosthenesON(){
224. FOR(i, 2, MAXN) {
225. if (! f[i]) primes[pc++] = i, f[i] = i;
226. for (int j = 0; j < pc && 1LL\*i\*primes[j] < MAXN && primes[j] <= f[i]; j++)
227. f[i\*primes[j]] = primes[j];
228. }
229. }
230. // \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*
232. // \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Criba de Atkin \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*
233. void Atkin() {
234. ll sqrtArraySize = sqrt(MAXN);
235. ll n;
237. bool isPrime[MAXN];
238. memset(isPrime, false, sizeof(isPrime));
240. ll pp[MAXN];
241. FOR(i, 0, sqrtArraySize + 5)    pp[i] = i \* i;
243. FOR(i, 1, sqrtArraySize + 1) {
244. FOR(j, 1, sqrtArraySize + 1) {
245. n = 4 \* pp[i] + pp[j];
247. if (n <= MAXN && (n % 12 == 1 || n % 12 == 5))
248. isPrime[n] = !isPrime[n];
250. n = 3 \* pp[i] + pp[j];
252. if (n <= MAXN && (n % 12 == 7))
253. isPrime[n] = !isPrime[n];
255. if (i > j) {
256. n = 3 \* pp[i] - pp[j];
257. if(n <= MAXN && n % 12 == 11)
258. isPrime[n] = !isPrime[n];
259. }
260. }
261. }
263. FOR(i, 5, sqrtArraySize + 1) {
264. if (isPrime[i]) {
265. ll omit = pp[i];
267. for (ll j = omit; j <= MAXN; j += omit) {
268. isPrime[j] = false;
269. }
270. }
271. }
273. if (MAXN >= 2) {
274. primes[pc ++] = 2;
275. if (MAXN >= 3) {
276. primes[pc ++] = 3;
277. }
278. }
280. FOR(i, 2, MAXN) {
281. if (isPrime[i]) {
282. primes[pc ++] = i;
283. }
284. }
285. }
286. // \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*
288. // \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Euler Totem \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*
289. // REGRESA PINCHE CANTIDAD DE COPRIMOS DE N
290. // Para sacar la puta suma de coprimos es n\*phi(n) / 2
291. ll eulerTotient(ll n) {
292. factorsOfN.clear();
293. getFactorsOfN(n);
294. FOR(i, 0, factorsOfN.size())
295. n = n / factorsOfN[i].first \* (factorsOfN[i].first - 1);
296. return n;
297. }

## Triangle of Mahonian

1. #define MAXN 35
2. #define MAXM 5005
4. ll arr[MAXN][MAXM];
6. int main() {
7. memset(arr, 0, sizeof(arr));
8. arr[0][0] = arr[1][0] = 1;
10. FOR(i, 0, MAXN) {
11. for(int m = 0; m < MAXM - 5 && arr[i][m] != 0; m ++) {
12. while(m < 5000 && ) {
13. FOR(j, 0, i+1) {
14. arr[i+1][j+m] += arr[i][m];
15. }
16. }
17. }
18. }
20. return 0;
21. }

# Segment Tree

## Lazy Propagation

1. const int maxN = 1e5, MOD = 1E9 + 7;
2. int HEIGHT, n;
3. ll d[maxN], t[maxN << 1];
5. void apply(int p, ll value) {
6. t[p] = value;
7. if (p < n) d[p] = value;
8. }
10. void build(int p) {
11. while (p > 1) p >>= 1, t[p] = max(max(t[p<<1], t[p<<1|1]), d[p]);
12. }
14. void push(int p) {
15. for (int s = HEIGHT; s > 0; --s) {
16. int i = p >> s;
17. if (d[i] != 0) {
18. apply(i<<1, d[i]);
19. apply(i<<1|1, d[i]);
20. d[i] = 0;
21. }
22. }
23. }
25. void change(int l, int r, ll value) {
26. l += n, r += n;
27. int l0 = l, r0 = r;
28. for (; l < r; l >>= 1, r >>= 1) {
29. if (l&1) apply(l++, value);
30. if (r&1) apply(--r, value);
31. }
32. build(l0);
33. build(r0 - 1);
34. }
36. ll query(int l, int r) {
37. l += n, r += n;
38. push(l);
39. push(r - 1);
40. ll res = LLONG\_MIN;
41. for (; l < r; l >>= 1, r >>= 1) {
42. if (l&1) res = max(res, t[l++]);
43. if (r&1) res = max(t[--r], res);
44. }
45. return res;
46. }

## Persistent Segment Tree

1. const int maxN = 1e6;
3. struct Node{
4. Node \*l, \*r;
5. int count;
6. Node(int count, Node \*l, Node \*r):count(count), l(l), r(r){}
7. Node \* insert(int, int, int);
8. };
9. Node \* segTrees[maxN + 1];
11. Node \* Node::insert(int left, int right, int pos){
12. if(pos < left || pos >= right)
13. return this;
14. if(left + 1 == right)
15. return new Node(count + 1, segTrees[0], segTrees[0]);
16. int m = (left + right) >> 1;
17. return new Node(count + 1, l -> insert(left, m, pos), r -> insert(m, right, pos));
18. }
19. int solve(Node \* last, Node \* first, int left, int right, int k){
20. if(left + 1 == right)
21. return left;
22. int m = (left + right) >> 1, dif = last -> l -> count - first -> l -> count;
23. if(dif < k)
24. return solve(last -> r, first -> r, m, right, k - dif);
25. return solve(last -> l, first -> l, left, m, k);
26. }
28. int main() {
29. ios::sync\_with\_stdio(false);
30. cin.tie(0);
31. cout.tie(0);
32. int n, q, x, y, k, arr[maxN], vals[maxN];
33. segTrees[0] = new Node(0, NULL, NULL);
34. segTrees[0] -> l = segTrees[0] -> r = segTrees[0];
35. while(cin >> n >> q){
36. map<int, int> myMap;
37. for(int i = 0; i < n; i++){
38. cin >> arr[i];
39. myMap[arr[i]];
40. }
41. int valSize = 0;
42. for(map<int, int>::iterator it = myMap.begin(); it != myMap.end(); it++){
43. it -> second = valSize;
44. vals[valSize++] = it -> first;
45. }
46. for(int i = 1; i <= n; i++)
47. segTrees[i] = segTrees[i - 1] -> insert(0, valSize, myMap[arr[i - 1]]);
48. while(q--){
49. cin >> x >> y >> k;
50. cout << vals[solve(segTrees[y], segTrees[x - 1], 0, valSize, k)] << '**\n**';
51. }
52. }
53. }

# Strings

## KMP

1. int lps[MAXN];
2. vi v;
4. void computeLPSArray(string pat, int M) {
5. int len = 0;
6. int i = 1;
7. lps[0] = 0;
8. while (i < M) {
9. if (pat[i] == pat[len])
10. lps[i ++] = ++ len;
11. else {
12. if (len != 0)   len = lps[len - 1];
13. else            lps[i ++] = 0;
14. }
15. }
16. }
17. void KMPSearch(string pat, string txt) {
18. int M = pat.length();
19. int N = txt.length();
20. int i = 0;
21. int j = 0;
23. computeLPSArray(pat, M);
25. while (i < N) {
26. if (pat[j] == txt[i]) {
27. j ++;
28. i ++;
29. }
31. if (j == M) {
32. v.pb(i - j);
33. j = lps[j - 1];
34. }
35. else if (i < N && pat[j] != txt[i]) {
36. if (j != 0)     j = lps[j - 1];
37. else            i = i + 1;
38. }
39. }
40. }
42. int main() {
43. string txt = "ABABDABACDABABCABAB";
44. string pat = "AB";
45. KMPSearch(pat, txt);
47. FOR(i, 0, v.size())
48. cout << v[i] << " **\n**"[i==v.size()-1];
50. return 0;
51. }

## Suffix Array

1. typedef pair<int, int> ii;
2. typedef pair<ii, int> iii;
3. typedef long long ll;
4. typedef vector<ll> vd;
5. typedef vector<vd> Matrix;
6. const int maxN = 1e6;
8. iii sa[maxN], aux[maxN];
9. int inv[maxN], lcp[maxN + 1], values[maxN + 1], n;
10. string st;
12. #define a(i) (fir? sa[i].FIRST: sa[i].SECOND)
14. void radiixSort(bool fir){
15. memset(values, 0, sizeof(values));
16. for(int i = 0; i < n; i++)
17. values[a(i) + 1]++;
18. for(int i = 1; i <= n; i++)
19. values[i] += values[i - 1];
20. for(int i = n - 1; i >= 0; i--)
21. aux[--values[a(i) + 1]] = sa[i];
22. for(int i = 0; i < n; i++)
23. sa[i] = aux[i];
24. }
26. void createSuffixArray(){
27. for(int i = 0; i < n; i++)
28. sa[i] = iii(ii(st[i], 0), i);
29. sort(sa, sa + n);
30. for(int cnt = 1; cnt <= n; cnt <<= 1){
31. for(int i = 0, j = 0; i < n; i = j)
32. for(ii cur = sa[i].first; j < n && sa[j].first == cur; j++)
33. sa[j].FIRST = inv[sa[j].THIRD] = i;
34. for(int i = 0; i < n; i++)
35. sa[i].SECOND = (sa[i].THIRD + cnt < n)? sa[inv[sa[i].THIRD + cnt]].FIRST: -1;
36. radiixSort(false);
37. radiixSort(true);
38. }
39. }
41. void createLCPArray(){
42. for(int i = 0; i < n; i++)
43. inv[sa[i].THIRD] = i;
44. for(int i = 0, k = 0; i < n; i++, k? k--: k){
45. if(inv[i] + 1 == n){
46. k = lcp[n - 1] = 0;
47. continue;
48. }
49. int cur = sa[inv[i]].THIRD, next = sa[inv[i] + 1].THIRD;
50. for(; max(cur, next) + k < n && st[cur + k] == st[next + k]; k++);
51. lcp[inv[i] + 1] = k;
52. }
53. }
55. int main() {
56. ios::sync\_with\_stdio(false), cin.tie(0), cout.tie(0);
57. cin >> n >> st;
58. createSuffixArray();
59. createLCPArray();
60. }

## Z Function

1. // Devuelve el arreglo Z
2. vector<int> z\_function(string &s){
3. int L = 0, R = 0,  n = s.length();
4. vector<int> z(n);
5. for(int i = 1; i < n; i++){
6. if(i <= R)
7. z[i] = min(z[i-L], R - i + 1);
8. while(i + z[i] < n && s[i + z[i]] == s[z[i]])
9. z[i]++;
10. if(i + z[i] - 1 > R){
11. L = i;
12. R = i + z[i] - 1;
13. }
14. }
15. return z;
16. }
18. int main(){ io
19. string A, B;
20. cin >> A >> B;
21. string z\_Arg = B + '$' + A;
22. vector<int> z = z\_function(z\_Arg);
23. }

# Variuos

## 2-Sat

1. #define MAXN 100005
2. #define MOD 1000000007
4. struct TwoSAT{
5. int n;
6. vector<int> G[MAXN\*2];
7. int S[MAXN\*2], c;
8. bool mark[MAXN\*2];
10. bool dfs(int x){
11. if(mark[x^1]) return false;
12. if(mark[x]) return true;
13. mark[x] = true;
14. S[c++] = x;
16. FOR(i, 0, G[x].size())
17. if(!dfs(G[x][i])) return false;
18. return true;
19. }
21. void init(int n){
22. this->n = n;
23. FOR(i, 0, 2\*n) G[i].clear();
24. memset(mark, 0, sizeof(mark));
25. }
27. */\**
28. *\* Each clause is in the form x or y*
29. *\* x is abs(x) - 1 and xval is x < 0 ? 1 : 0*
30. *\*/*
31. void add\_clause(int x, int xval, int y, int yval){
32. x = x\*2 + xval;
33. y = y\*2 + yval;
34. G[x^1].push\_back(y);
35. G[y^1].push\_back(x);
36. }
38. bool solve(){
39. for(int i = 0; i < 2\*n; i += 2){
40. if(!mark[i] && !mark[i+1]){
41. c = 0;
42. if(!dfs(i)){
43. while(c > 0) mark[S[--c]] = false;
44. if(!dfs(i + 1)) return false;
45. }
46. }
47. }
48. return true;
49. }
50. }TS;
52. int main(){
53. int n, m;
54. while(scanf("%d %d", &n, &m) != EOF){
55. TS.init(n);
56. // scan m clauses
57. FOR(i, 0, m){
58. int a, b;
59. scanf("%d %d", &a, &b);
60. TS.add\_clause(abs(a) - 1, a < 0 ? 1 : 0, abs(b) - 1, b < 0 ? 1 : 0);
61. }
62. printf("%d**\n**", TS.solve()? 1 : 0);
63. }
64. return 0;
65. }

## Bits

1. typedef unsigned int ui;
2. typedef unsigned long long ull;
4. ull BitCount(ull u) {
5. ull uCount = u - ((u >> 1) & 033333333333) - ((u >> 2) & 011111111111);
6. return ((uCount + (uCount >> 3)) & 030707070707) % 63;
7. }
9. ull flipBits(ull n, int k) {
10. ull mask = (1 << k) - 1;
11. return ~n & mask;
12. }
14. ull flipBits(ull n) {
15. return ~n;
16. }
18. ull differentBits(ull a, ull b) {
19. return BitCount(a ^ b);
20. }
22. void getEvenOddBits(ull n) {
23. // Para ui, hacerlo con 8 A's o 5's
24. ull evenBits = n & 0xAAAAAAAAAAAAAAAA;
25. ull oddBits = n & 0x5555555555555555;
26. }
28. ull rotateBits(ull n, int d) {
29. // d negative for left rotation, positive for right.
30. d %= 64;
31. return (n >> d) | (n << (64 - d));
32. }
34. string toBinary(ull n) {
35. string s = "";
36. while(n) {
37. if (n & 1)  s = "1" + s;
38. else        s = "0" + s;
39. n >>= 1;
40. }
41. return s;
42. }
44. ui getSetBitsFromOneToN(ull n){
45. ui two = 2, ans = 0;
46. ull N = n;
47. while(n) {
48. ans += (N / two) \* (two >> 1);
49. if ((N & (two - 1)) > (two >> 1) - 1)
50. ans += (N & (two - 1)) - (two >> 1) + 1;
51. two <<= 1;
52. n >>= 1;
53. }
54. return ans;
55. }

## Fast IO

1. const int BUFFSIZE = 10240;
2. char BUFF[BUFFSIZE + 1], \*ppp = BUFF;
3. int RR, CHAR, SIGN, BYTES = 0;
4. #define GETCHAR(c) { \
5. if(ppp-BUFF==BYTES && (BYTES==0 || BYTES==BUFFSIZE)) { BYTES = fread(BUFF,1,BUFFSIZE,stdin); ppp=BUFF; } \
6. if(ppp-BUFF==BYTES && (BYTES>0 && BYTES<BUFFSIZE)) { BUFF[0] = 0; ppp=BUFF; }\
7. c = \*ppp++; \
8. }
9. #define DIGIT(c) (((c) >= ’0’) && ((c) <= ’9’))
10. #define MINUS(c) ((c)== ’-’)
11. #define GETNUMBER(n) { \
12. n = 0; SIGN = 1; do { GETCHAR(CHAR); } while(!(DIGIT(CHAR) || MINUS(CHAR))); \
13. if(MINUS(CHAR)) { SIGN = -1; GETCHAR(CHAR); } \
14. while(DIGIT(CHAR)) { n = ((n<<3) + (n << 1)) + CHAR-’0’; GETCHAR(CHAR); } if(SIGN == -1) { n = -n; } \
15. }

## FFT

1. struct Complex{
2. double real, imag;
3. Complex(){}
4. Complex(const complex<double> &a): real(a.real()), imag(a.imag()){}
5. Complex(double real, double imag): real(real), imag(imag){}
6. Complex operator\*(const Complex & a){
7. return Complex(real \* a.real - imag \* a.imag, real \* a.imag + a.real \* imag);
8. }
9. Complex operator+(const Complex & a){return Complex(real + a.real, imag + a.imag);}
10. };
11. Complex conj(const Complex & a){return Complex(a.real, -a.imag);}
13. const int maxN = 1e6;
14. int maxL;
16. int reverses[maxN << 2];
17. bool v[maxN << 2];
18. Complex steps[30], unityRoots[maxN << 2], first[maxN << 2], second[maxN << 2];
19. ll sums[maxN << 2];
21. int reverse(int num, int exp){
22. int reverse = 0, pot = 1, inv = 1 << (exp - 1);
23. while(inv >= 1){
24. if(num & pot) reverse |= inv;
25. inv >>= 1, pot <<= 1;
26. }
27. return reverse;
28. }
30. void sqRoot(Complex \*vec, int len, int loog){
31. for(int i = len - 1; i >= 0; i--){
32. vec[i << 1] = vec[i];
33. vec[(i << 1) + 1] = vec[i] \* steps[loog];
34. }
35. }
37. void FFT(Complex \*coef, int arrS){
38. const int saiz = ceil(log2(arrS));
40. memset(v, false, sizeof(v));
41. for(int i = 0; i < (1 << saiz); i++)
42. if(!v[reverses[i] >> (maxL - saiz)])
43. swap(coef[i], coef[reverses[i] >> (maxL - saiz)]), v[i] = true;
44. unityRoots[0] = Complex(1, 0);
46. //DIVIDE AND CONQUER...
47. for(int T = 1, u = 0; T < (1 << saiz); T <<= 1, u++){
48. sqRoot(unityRoots, T, u);
49. for(int i = 0; i < (1 << saiz); i += (T << 1)){
50. for(int j = 0; j < T; j++){
51. Complex lTmp = coef[i + j], rTmp = coef[i + j + T];
53. coef[i + j] = lTmp + rTmp \* unityRoots[j];
54. coef[i + j + T] = lTmp + rTmp \* unityRoots[j + T];
55. }
56. }
57. }
58. }
60. //one & two = coefficient arrays w/sizes, res is where results are stored
61. void polynomMultiplication(int \*one, int \*two, int \*res, int oS, int tS){
62. const int saiz = 1 << int(ceil(log2(oS + tS)));
64. for(int i = 0; i < saiz; i++)
65. first[i] = Complex(i < oS? double(one[i]): 0.0, 0.0);
66. for(int i = 0; i < saiz; i++)
67. second[i] = Complex(i < tS? double(two[i]): 0.0, 0.0);
69. //FFT
70. FFT(first, saiz), FFT(second, saiz);
72. //INVERSE FFT = FFT(conj(C1 \* C2) / N)
73. for(int i = 0; i < saiz; i++)
74. first[i] = conj(first[i] \* second[i]);
75. FFT(first, saiz);
77. for(int i = 0; i < saiz; i++)
78. res[i] = int(round(first[i].real / saiz));
79. }

## Fraction

1. struct Fraction {
2. ll num, den;
4. Fraction() {num = den = 1;}
6. Fraction(ll numm, ll denn) : num(numm), den(denn) {
7. simplify();
8. }
10. void simplify() {
11. bool signo = false;
12. if (num < 0)    signo = !signo;
13. if (den < 0)    signo = !signo;
14. ll g = \_\_gcd(num, den);
15. if (g) {
16. num /= g;
17. den /= g;
18. }
19. if (signo)  num = -num;
20. }
22. Fraction operator + (const Fraction &r) const {
23. Fraction ret;
24. ret.num = num\*r.den + r.num\*den;
25. ret.den = r.den\*den;
26. ret.simplify();
27. return ret;
28. }
30. Fraction operator + (const ll &r) const {
31. Fraction ret;
32. ret.num = num + r\*den;
33. ret.den = den;
34. return ret;
35. }
37. Fraction operator - (const Fraction &r) const {
38. Fraction ret;
39. ret.num = num\*r.den - r.num\*den;
40. ret.den = r.den\*den;
41. ret.simplify();
42. return ret;
43. }
45. Fraction operator - (const ll &r) const {
46. Fraction ret;
47. ret.num = num - r \* den;
48. ret.den = den;
49. return ret;
50. }
52. Fraction operator \* (const Fraction &r) const {
53. Fraction ret;
54. ret.num = num\*r.num;
55. ret.den = den\*r.den;
56. ret.simplify();
57. return ret;
58. }
60. Fraction operator \* (const ll &r) const {
61. Fraction ret;
62. ret.num = num\*r;
63. ret.den = den;
64. ret.simplify();
65. return ret;
66. }
68. Fraction operator / (const Fraction &r) const {
69. Fraction ret;
70. ret.num = num\*r.den;
71. ret.den = den\*r.num;
72. ret.simplify();
73. return ret;
74. }
76. Fraction operator / (const ll &r) const {
77. Fraction ret;
78. ret.num = num;
79. ret.den = den\*r;
80. ret.simplify();
81. return ret;
82. }
84. Fraction pow(int n) const {
85. Fraction ret(1, 1);
86. Fraction x(num, den);
87. while (n) {
88. if (n & 1) ret = ret \* x;
89. x = x \* x;
90. n >>= 1;
91. }
92. return ret;
93. }
95. bool operator <(const Fraction &r) const {
96. return num \* r.den < den \* r.num;
97. }
99. bool operator ==(const Fraction &r) const {
100. return num == r. num && den == r.den;
101. }
103. string to\_str() {
104. return to\_string(num) + "/" + to\_string(den);
105. }
106. };
108. int main() { \_
109. Fraction f;
110. while(cin >> f.num >> f.den) {
111. f.simplify();
112. cout << f.num << "/" << f.den << endl;
113. }
115. return 0;
116. }

## Hour

1. struct Hora {
2. int h, m, s, t;
4. Hora() { t = h = m = s = 0; }
6. Hora(int hh, int mm, int ss) : h(hh), m(mm), s(ss) {
7. simplify();
8. }
10. Hora(string ss) {
11. int f1 = ss.find(":");
12. h = atoi(ss.substr(0, f1).c\_str());
14. int f2 = ss.find(":", f1+1);
15. if (f2 != -1) {
16. m = atoi(ss.substr(f1+1, f2-f1).c\_str());
17. s = atoi(ss.substr(f2+1).c\_str());
18. }
19. else {
20. m = atoi(ss.substr(f1+1).c\_str());
21. s = 0;
22. }
24. simplify();
25. }
27. Hora operator +(const Hora &r) const {
28. Hora ret;
30. ret.h = h + r.h;
31. ret.m = m + r.m;
32. ret.s = s + r.s;
34. ret.simplify();
36. return ret;
37. }
39. Hora operator -(const Hora &r) const {
40. Hora ret;
42. ret.h = h - r.h;
43. ret.m = m - r.m;
44. ret.s = s - r.s;
46. ret.simplify();
48. return ret;
49. }
51. bool operator <(const Hora &r) const {
52. return t < r.t;
53. }
55. bool operator ==(const Hora &r) const {
56. return t == r.t;
57. }
59. void simplify() {
60. t = 3600 \* h + 60 \* m + s;
61. t = (t + 86400) % 86400;
63. int tt = t;
64. h = tt / 3600;
65. tt %= 3600;
66. m = tt / 60;
67. s = tt % 60;
68. }
70. string to\_str() {
71. string ret = "";
73. if (h < 10) ret = "0";
74. ret += std::to\_string(h) + ":";
76. if (m < 10) ret += "0";
77. ret += std::to\_string(m) + ":";
79. if (s < 10) ret += "0";
80. ret += std::to\_string(s);
82. return ret;
83. }
85. string to\_str(const bool b) {
86. string ret = "";
88. if (h < 10) ret = "0";
89. ret += to\_string(h) + ":";
91. if (m < 10) ret += "0";
92. ret += to\_string(m);
94. return ret;
95. }
96. };
98. int main() { \_
99. Hora h1, h2;
100. string s1, s2;
102. while(cin >> s1 >> s2) {
103. h1 = Hora(s1);
104. h2 = Hora(s2);
106. cout << (h1 - h2).to\_str() << endl;
107. }
109. return 0;
110. }

## Matrix Exponentiation

1. #define FIRST first
2. #define SECOND second.first
3. #define THIRD second.second
5. using namespace std;
6. typedef pair<int, int> ii;
7. typedef vector<double> vd;
8. typedef vector<vd> Matrix;
9. typedef long long ll;
11. const int maxN = 100;
13. Matrix operator \*(const Matrix &first, const Matrix &second){
14. Matrix ret(first.size(), vd(second[0].size(), 0.0));
15. for(int i = 0; i < first.size(); i++)
16. for(int j = 0; j < second[0].size(); j++)
17. for(int k = 0; k < second.size(); k++)
18. ret[i][j] += first[i][k] \* second[k][j];
19. return ret;
20. }
21. Matrix operator^(Matrix mat, int coef){
22. Matrix ret(mat.size(), vd(mat.size(), 0.0));
23. for(int i = 0; i < mat.size(); i++) ret[i][i] = 1.0;
24. while(coef){
25. if(coef & 1)
26. ret = ret \* mat;
27. coef >>= 1, mat = mat \* mat;
28. }
29. return ret;
30. }

## Roman Numbers

1. #define MAXN 10
2. #define MOD 1000000007
4. int mil[MAXN];
6. string fill(char c, int n) {
7. string s;
8. while (n--) s += c;
9. return s;
10. }
12. string toRoman(ll n, int nivel) {
13. if (n == 0) return "";
14. if (n < 4) return fill('I', n);
15. if (n < 6) return fill('I', 5 - n) + "V";
16. if (n < 9) return string("V") + fill('I', n - 5);
17. if (n < 11) return fill('I', 10 - n) + "X";
18. if (n < 40) return fill('X', n / 10) + toRoman(n % 10, nivel);
19. if (n < 60) return fill('X', 5 - n / 10) + 'L' + toRoman(n % 10, nivel);
20. if (n < 90) return string("L") + fill('X', n / 10 - 5 ) + toRoman(n % 10, nivel);
21. if (n < 110) return fill('X', 10 - n / 10) + "C" + toRoman(n % 10, nivel);
22. if (n < 400) return fill('C', n / 100) + toRoman(n % 100, nivel);
23. if (n < 600) return fill('C', 5 - n / 100) + 'D' + toRoman(n % 100, nivel);
24. if (n < 900) return string("D" ) + fill('C', n / 100 - 5) + toRoman(n % 10, nivel);
25. if (n < 1100) return fill('C', 10 - n / 100) + "M" + toRoman(n % 100, nivel);
26. if (n < 4000) return fill('M', n / 1000) + toRoman(n % 1000, nivel);
28. string ret = toRoman(n / 1000, nivel + 1);
29. string ret2 = toRoman(n % 1000, nivel);
30. mil[nivel] = ret.length();
31. if (ret2 == "")    return ret;
32. return ret + " " + toRoman(n % 1000, nivel);
33. }
35. string toRoman(ll n) {
36. string ret = toRoman(n, 0);
37. for(int i = 0; mil[i]; i ++) {
38. ret = fill('\_', mil[i]) + "**\n**" + ret;
39. }
41. return ret;
42. }
44. int main() { \_
45. ll n;
46. while (cin >> n) {
47. cout << toRoman(n) << endl;
48. }
49. return 0;
50. }

## Rotate Matrix

1. #define MAXN 10
2. #define MOD 1000000007
4. int mat[MAXN][MAXN];
6. void giraMat(int k, int n) {
7. k %= 4;
9. FOR(p, 0, k) {
10. FOR(i, 0, n/2) {
11. FOR(j, i, n - i - 1) {
12. ii p1(i, j);
13. ii p2(j, n - i - 1);
14. ii p3(n - i - 1, n - j - 1);
15. ii p4(n - j - 1, i);
17. int aux = mat[p1.first][p1.second];
18. mat[p1.first][p1.second] = mat[p4.first][p4.second];
19. mat[p4.first][p4.second] = mat[p3.first][p3.second];
20. mat[p3.first][p3.second] = mat[p2.first][p2.second];
21. mat[p2.first][p2.second] = aux;
22. }
23. }
24. }
25. }
27. int main() { \_
28. int cant = 1;
29. FOR(i, 0, MAXN) FOR(j, 0, MAXN) {
30. mat[i][j] = cant;
31. cant ++;
32. }
34. giraMat(6, MAXN);
36. FOR(i, 0, MAXN) FOR(j, 0, MAXN)
37. cout << mat[i][j] << "**\t\n**"[j == MAXN-1];
38. return 0;
39. }

## Sorts

1. #define MAXN 15
2. #define MOD 1000000007
4. int arr[MAXN];
5. int comp, inter;
7. void print() {
8. FOR(i, 0, MAXN) {
9. cout << arr[i] << " **\n**"[i == MAXN-1];
10. }
11. cout << endl;
12. }
14. void bubbleSort() {
15. int n = MAXN;
16. bool b = true;
18. for(int i = 0; i < n - 1 && b; i ++){
19. b = false;
20. FOR(j, 0, n - 1 - i) {
21. if (arr[j+1] < arr[j]) {
22. swap(arr[j], arr[j + 1]);
23. b = true;
25. inter ++;
26. }
27. comp ++;
28. }
29. }
30. }
32. void selectionSort() {
33. int n = MAXN;
34. FOR(i, 0, n - 1) {
35. int mini = i;
36. FOR(j, i + 1, n) {
37. comp ++;
38. if (arr[j] < arr[mini]) mini = j;
40. }
42. if (i != mini) {
43. swap(arr[i], arr[mini]);
44. inter ++;
45. }
46. }
47. }
49. void intercambioSort() {
50. int n = MAXN;
51. FOR(i, 0, n - 1) {
52. FOR(j, i + 1, n) {
53. if (arr[i] > arr[j]) {
54. swap(arr[i], arr[j]);
55. inter++;
56. }
57. comp++;
58. }
59. }
60. }
62. void insertionSort() {
63. int j, key, n = MAXN;
64. FOR(i, 1, n) {
65. key = arr[i];
66. j = i - 1;
67. while (j >= 0 && arr[j] > key) {
68. inter ++;
69. arr[j+1] = arr[j];
70. j = j - 1;
71. }
73. comp = inter;
74. if(j)   comp ++;
76. arr[j+1] = key;
77. }
78. }
80. void insertionBinarySort() {
81. int n = MAXN;
82. vi v;
83. FOR(i, 0, n) {
84. v.insert(upper\_bound(v.begin(), v.end(), arr[i]), arr[i]);
85. }
87. copy(v.begin(), v.end(), arr);
88. }

91. void Une(int ini, int mitad, int fin) {
92. int aux[fin - ini + 1];
93. int i = ini, j = mitad + 1, k = 0;
95. while(i <= mitad && j <= fin) {
96. if (arr[i] < arr[j]) {
97. aux[k] = arr[i];
98. i ++;
99. }
100. else {
101. aux[k] = arr[j];
102. j ++;
103. }
104. k ++;
105. }
107. for(; i <= mitad; i ++, k ++)    aux[k] = arr[i];
108. for(; j <= fin; j ++, k ++)      aux[k] = arr[j];
110. for(k = 0; ini <= fin; ini ++, k ++)
111. arr[ini] = aux[k];
112. }
114. void mergeSort(int ini, int fin){
115. if(ini < fin){
116. int mitad = (ini + fin) / 2;
117. mergeSort(ini, mitad);
118. mergeSort(mitad + 1, fin);
119. Une(ini, mitad, fin);
120. }
121. }

124. void heapify(int n, int i) {
125. int maxi = i;
126. int l = 2\*i + 1;
127. int r = 2\*i + 2;
129. if (l < n && arr[l] > arr[maxi])
130. maxi = l;
132. if (r < n && arr[r] > arr[maxi])
133. maxi = r;
135. if (maxi != i) {
136. swap(arr[i], arr[maxi]);
137. heapify(n, maxi);
138. }
139. }
141. void heapSort() {
142. int n = MAXN;
143. for (int i = n / 2 - 1; i >= 0; i--)
144. heapify(n, i);
146. for (int i = n-1; i >= 0; i--) {
147. swap(arr[0], arr[i]);
148. heapify(i, 0);
149. }
150. }
152. int part(int low, int high) {
153. int pivot = arr[high];
154. int i = low - 1;
156. FOR(j, low, high) {
157. if (arr[j] <= pivot) {
158. i++;
159. swap(arr[i], arr[j]);
160. }
161. }
162. swap(arr[i + 1], arr[high]);
163. return (i + 1);
164. }
166. // QuickSort with less recursion calls.
167. void quickSortNoTail(int low, int high) {
168. while (low < high) {
169. int pi = part(low, high);
171. if (pi - low < high - pi) {
172. quickSortNoTail(low, pi - 1);
173. low = pi + 1;
174. }
175. else {
176. quickSortNoTail(pi + 1, high);
177. high = pi - 1;
178. }
179. }
180. }
182. void quickSort(int low, int high) {
183. if (low < high) {
184. int pi = part(low, high);
185. quickSort(low, pi - 1);
186. quickSort(pi + 1, high);
187. }
188. }

# Otros

## Series

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

## First 5000 digits of PI

3.1415926535897932384626433832795028841971693993751058209749445923078164062862089986280348253421170679821480865132823066470938446095505822317253594081284811174502841027019385211055596446229489549303819644288109756659334461284756482337867831652712019091456485669234603486104543266482133936072602491412737245870066063155881748815209209628292540917153643678925903600113305305488204665213841469519415116094330572703657595919530921861173819326117931051185480744623799627495673518857527248912279381830119491298336733624406566430860213949463952247371907021798609437027705392171762931767523846748184676694051320005681271452635608277857713427577896091736371787214684409012249534301465495853710507922796892589235420199561121290219608640344181598136297747713099605187072113499999983729780499510597317328160963185950244594553469083026425223082533446850352619311881710100031378387528865875332083814206171776691473035982534904287554687311595628638823537875937519577818577805321712268066130019278766111959092164201989380952572010654858632788659361533818279682303019520353018529689957736225994138912497217752834791315155748572424541506959508295331168617278558890750983817546374649393192550604009277016711390098488240128583616035637076601047101819429555961989467678374494482553797747268471040475346462080466842590694912933136770289891521047521620569660240580381501935112533824300355876402474964732639141992726042699227967823547816360093417216412199245863150302861829745557067498385054945885869269956909272107975093029553211653449872027559602364806654991198818347977535663698074265425278625518184175746728909777727938000816470600161452491921732172147723501414419735685481613611573525521334757418494684385233239073941433345477624168625189835694855620992192221842725502542568876717904946016534668049886272327917860857843838279679766814541009538837863609506800642251252051173929848960841284886269456042419652850222106611863067442786220391949450471237137869609563643719172874677646575739624138908658326459958133904780275900994657640789512694683983525957098258226205224894077267194782684826014769909026401363944374553050682034962524517493996514314298091906592509372216964615157098583874105978859597729754989301617539284681382686838689427741559918559252459539594310499725246808459872736446958486538367362226260991246080512438843904512441365497627807977156914359977001296160894416948685558484063534220722258284886481584560285060168427394522674676788952521385225499546667278239864565961163548862305774564980355936345681743241125150760694794510965960940252288797108931456691368672287489405601015033086179286809208747609178249385890097149096759852613655497818931297848216829989487226588048575640142704775551323796414515237462343645428584447952658678210511413547357395231134271661021359695362314429524849371871101457654035902799344037420073105785390621983874478084784896833214457138687519435064302184531910484810053706146806749192781911979399520614196634287544406437451237181921799983910159195618146751426912397489409071864942319615679452080951465502252316038819301420937621378559566389377870830390697920773467221825625996615014215030680384477345492026054146659252014974428507325186660021324340881907104863317346496514539057962685610055081066587969981635747363840525714591028970641401109712062804390397595156771577004203378699360072305587631763594218731251471205329281918261861258673215791984148488291644706095752706957220917567116722910981690915280173506712748583222871835209353965725121083579151369882091444210067510334671103141267111369908658516398315019701651511685171437657618351556508849099898599823873455283316355076479185358932261854896321329330898570642046752590709154814165498594616371802709819943099244889575712828905923233260972997120844335732654893823911932597463667305836041428138830320382490375898524374417029132765618093773444030707469211201913020330380197621101100449293215160842444859637669838952286847831235526582131449576857262433441893039686426243410773226978028073189154411010446823252716201052652272111660396665573092547110557853763466820653109896526918620564769312570586356620185581007293606598764861179104533488503461136576867532494416680396265797877185560845529654126654085306143444318586769751456614068007002378776591344017127494704205622305389945613140711270004078547332699390814546646458807972708266830634328587856983052358089330657574067954571637752542021149557615814002501262285941302164715509792592309907965473761255176567513575178296664547791745011299614890304639947132962107340437518957359614589019389713111790429782856475032031986915140287080859904801094121472213179476477726224142548545403321571853061422881375850430633217518297986622371721591607716692547487389866549494501146540628433663937900397692656721463853067360965712091807638327166416274888800786925602902284721040317211860820419000422966171196377921337575114959501566049631862947265473642523081770367515906735023507283540567040386743513622224771589150495309844489333096340878076932599397805419341447377441842631298608099888687413260472

## First 150 Fibonacci numbers

0 : 0

1 : 1

2 : 1

3 : 2

4 : 3

5 : 5

6 : 8 = 23

7 : 13

8 : 21 = 3 x 7

9 : 34 = 2 x 17

10 : 55 = 5 x 11

11 : 89

12 : 144 = 24 x 32

13 : 233

14 : 377 = 13 x 29

15 : 610 = 2 x 5 x 61

16 : 987 = 3 x 7 x 47

17 : 1597

18 : 2584 = 23 x 17 x 19

19 : 4181 = 37 x 113

20 : 6765 = 3 x 5 x 11 x 41

21 : 10946 = 2 x 13 x 421

22 : 17711 = 89 x 199

23 : 28657

24 : 46368 = 25 x 32 x 7 x 23

25 : 75025 = 52 x 3001

26 : 121393 = 233 x 521

27 : 196418 = 2 x 17 x 53 x 109

28 : 317811 = 3 x 13 x 29 x 281

29 : 514229

30 : 832040 = 23 x 5 x 11 x 31 x 61

31 : 1346269 = 557 x 2417

32 : 2178309 = 3 x 7 x 47 x 2207

33 : 3524578 = 2 x 89 x 19801

34 : 5702887 = 1597 x 3571

35 : 9227465 = 5 x 13 x 141961

36 : 14930352 = 24 x 33 x 17 x 19 x 107

37 : 24157817 = 73 x 149 x 2221

38 : 39088169 = 37 x 113 x 9349

39 : 63245986 = 2 x 233 x 135721

40 : 102334155 = 3 x 5 x 7 x 11 x 41 x 2161

41 : 165580141 = 2789 x 59369

42 : 267914296 = 23 x 13 x 29 x 211 x 421

43 : 433494437

44 : 701408733 = 3 x 43 x 89 x 199 x 307

45 : 1134903170 = 2 x 5 x 17 x 61 x 109441

46 : 1836311903 = 139 x 461 x 28657

47 : 2971215073

48 : 4807526976 = 26 x 32 x 7 x 23 x 47 x 1103

49 : 7778742049 = 13 x 97 x 6168709

50 : 12586269025 = 52 x 11 x 101 x 151 x 3001

51 : 20365011074 = 2 x 1597 x 6376021

52 : 32951280099 = 3 x 233 x 521 x 90481

53 : 53316291173 = 953 x 55945741

54 : 86267571272 = 23 x 17 x 19 x 53 x 109 x 5779

55 : 139583862445 = 5 x 89 x 661 x 474541

56 : 225851433717 = 3 x 72 x 13 x 29 x 281 x 14503

57 : 365435296162 = 2 x 37 x 113 x 797 x 54833

58 : 591286729879 = 59 x 19489 x 514229

59 : 956722026041 = 353 x 2710260697

60 : 1548008755920 = 24 x 32 x 5 x 11 x 31 x 41 x 61 x 2521

61 : 2504730781961 = 4513 x 555003497

62 : 4052739537881 = 557 x 2417 x 3010349

63 : 6557470319842 = 2 x 13 x 17 x 421 x 35239681

64 : 10610209857723 = 3 x 7 x 47 x 1087 x 2207 x 4481

65 : 17167680177565 = 5 x 233 x 14736206161

66 : 27777890035288 = 23 x 89 x 199 x 9901 x 19801

67 : 44945570212853 = 269 x 116849 x 1429913

68 : 72723460248141 = 3 x 67 x 1597 x 3571 x 63443

69 : 117669030460994 = 2 x 137 x 829 x 18077 x 28657

70 : 190392490709135 = 5 x 11 x 13 x 29 x 71 x 911 x 141961

71 : 308061521170129 = 6673 x 46165371073

72 : 498454011879264 = 25 x 33 x 7 x 17 x 19 x 23 x 107 x 103681

73 : 806515533049393 = 9375829 x 86020717

74 : 1304969544928657 = 73 x 149 x 2221 x 54018521

75 : 2111485077978050 = 2 x 52 x 61 x 3001 x 230686501

76 : 3416454622906707 = 3 x 37 x 113 x 9349 x 29134601

77 : 5527939700884757 = 13 x 89 x 988681 x 4832521

78 : 8944394323791464 = 23 x 79 x 233 x 521 x 859 x 135721

79 : 14472334024676221 = 157 x 92180471494753

80 : 23416728348467685 = 3 x 5 x 7 x 11 x 41 x 47 x 1601 x 2161 x 3041

81 : 37889062373143906 = 2 x 17 x 53 x 109 x 2269 x 4373 x 19441

82 : 61305790721611591 = 2789 x 59369 x 370248451

83 : 99194853094755497

84 : 160500643816367088 = 24 x 32 x 13 x 29 x 83 x 211 x 281 x 421 x 1427

85 : 259695496911122585 = 5 x 1597 x 9521 x 3415914041

86 : 420196140727489673 = 6709 x 144481 x 433494437

87 : 679891637638612258 = 2 x 173 x 514229 x 3821263937

88 : 1100087778366101931 = 3 x 7 x 43 x 89 x 199 x 263 x 307 x 881 x 967

89 : 1779979416004714189 = 1069 x 1665088321800481

90 : 2880067194370816120 = 23 x 5 x 11 x 17 x 19 x 31 x 61 x 181 x 541 x 109441

91 : 4660046610375530309 = 132 x 233 x 741469 x 159607993

92 : 7540113804746346429 = 3 x 139 x 461 x 4969 x 28657 x 275449

93 : 12200160415121876738 = 2 x 557 x 2417 x 4531100550901

94 : 19740274219868223167 = 2971215073 x 6643838879

95 : 31940434634990099905 = 5 x 37 x 113 x 761 x 29641 x 67735001

96 : 51680708854858323072 = 27 x 32 x 7 x 23 x 47 x 769 x 1103 x 2207 x 3167

97 : 83621143489848422977 = 193 x 389 x 3084989 x 361040209

98 : 135301852344706746049 = 13 x 29 x 97 x 6168709 x 599786069

99 : 218922995834555169026 = 2 x 17 x 89 x 197 x 19801 x 18546805133

100 : 354224848179261915075 = 3 x 52 x 11 x 41 x 101 x 151 x 401 x 3001 x 570601

101 : 573147844013817084101 = 743519377 x 770857978613

102 : 927372692193078999176 = 23 x 919 x 1597 x 3469 x 3571 x 6376021

103 : 1500520536206896083277 = 519121 x 5644193 x 512119709

104 : 2427893228399975082453 = 3 x 7 x 103 x 233 x 521 x 90481 x 102193207

105 : 3928413764606871165730 = 2 x 5 x 13 x 61 x 421 x 141961 x 8288823481

106 : 6356306993006846248183 = 953 x 55945741 x 119218851371

107 : 10284720757613717413913 = 1247833 x 8242065050061761

108 : 16641027750620563662096 = 24 x 34 x 17 x 19 x 53 x 107 x 109 x 5779 x 11128427

109 : 26925748508234281076009 = 827728777 x 32529675488417

110 : 43566776258854844738105 = 5 x 112 x 89 x 199 x 331 x 661 x 39161 x 474541

111 : 70492524767089125814114 = 2 x 73 x 149 x 2221 x 1459000305513721

112 : 114059301025943970552219 = 3 x 72 x 13 x 29 x 47 x 281 x 14503 x 10745088481

113 : 184551825793033096366333 = 677 x 272602401466814027129

114 : 298611126818977066918552 = 23 x 37 x 113 x 229 x 797 x 9349 x 54833 x 95419

115 : 483162952612010163284885 = 5 x 1381 x 28657 x 2441738887963981

116 : 781774079430987230203437 = 3 x 59 x 347 x 19489 x 514229 x 1270083883

117 : 1264937032042997393488322 = 2 x 17 x 233 x 29717 x 135721 x 39589685693

118 : 2046711111473984623691759 = 353 x 709 x 8969 x 336419 x 2710260697

119 : 3311648143516982017180081 = 13 x 1597 x 159512939815855788121

120 : 5358359254990966640871840 = 25 x 32 x 5 x 7 x 11 x 23 x 31 x 41 x 61 x 241 x 2161 x 2521 x 20641

121 : 8670007398507948658051921 = 89 x 97415813466381445596089

122 : 14028366653498915298923761 = 4513 x 555003497 x 5600748293801

123 : 22698374052006863956975682 = 2 x 2789 x 59369 x 68541957733949701

124 : 36726740705505779255899443 = 3 x 557 x 2417 x 3010349 x 3020733700601

125 : 59425114757512643212875125 = 53 x 3001 x 158414167964045700001

126 : 96151855463018422468774568 = 23 x 13 x 17 x 19 x 29 x 211 x 421 x 1009 x 31249 x 35239681

127 : 155576970220531065681649693 = 27941 x 5568053048227732210073

128 : 251728825683549488150424261 = 3 x 7 x 47 x 127 x 1087 x 2207 x 4481 x 186812208641

129 : 407305795904080553832073954 = 2 x 257 x 5417 x 8513 x 39639893 x 433494437

130 : 659034621587630041982498215 = 5 x 11 x 131 x 233 x 521 x 2081 x 24571 x 14736206161

131 : 1066340417491710595814572169

132 : 1725375039079340637797070384 = 24 x 32 x 43 x 89 x 199 x 307 x 9901 x 19801 x 261399601

133 : 2791715456571051233611642553 = 13 x 37 x 113 x 3457 x 42293 x 351301301942501

134 : 4517090495650391871408712937 = 269 x 4021 x 116849 x 1429913 x 24994118449

135 : 7308805952221443105020355490 = 2 x 5 x 17 x 53 x 61 x 109 x 109441 x 1114769954367361

136 : 11825896447871834976429068427 = 3 x 7 x 67 x 1597 x 3571 x 63443 x 23230657239121

137 : 19134702400093278081449423917

138 : 30960598847965113057878492344 = 23 x 137 x 139 x 461 x 691 x 829 x 18077 x 28657 x 1485571

139 : 50095301248058391139327916261 = 277 x 2114537501 x 85526722937689093

140 : 81055900096023504197206408605 = 3 x 5 x 11 x 13 x 29 x 41 x 71 x 281 x 911 x 141961 x 12317523121

141 : 131151201344081895336534324866 = 2 x 108289 x 1435097 x 142017737 x 2971215073

142 : 212207101440105399533740733471 = 6673 x 46165371073 x 688846502588399

143 : 343358302784187294870275058337 = 89 x 233 x 8581 x 1929584153756850496621

144 : 555565404224292694404015791808 = 26 x 33 x 7 x 17 x 19 x 23 x 47 x 107 x 1103 x 103681 x 10749957121

145 : 898923707008479989274290850145 = 5 x 514229 x 349619996930737079890201

146 : 1454489111232772683678306641953 = 151549 x 9375829 x 86020717 x 11899937029

147 : 2353412818241252672952597492098 = 2 x 13 x 97 x 293 x 421 x 3529 x 6168709 x 347502052673

148 : 3807901929474025356630904134051 = 3 x 73 x 149 x 2221 x 11987 x 54018521 x 81143477963

149 : 6161314747715278029583501626149 = 110557 x 162709 x 4000949 x 85607646594577

150 : 9969216677189303386214405760200 = 23 x 52 x 11 x 31 x 61 x 101 x 151 x 3001 x 12301 x 18451 x 23068650

## First 1000 prime numbers

2 3 5 7 11 13 17 19 23 29 31 37 41 43 47

53 59 61 67 71 73 79 83 89 97 101 103 107 109 113

127 131 137 139 149 151 157 163 167 173 179 181 191 193 197

199 211 223 227 229 233 239 241 251 257 263 269 271 277 281

283 293 307 311 313 317 331 337 347 349 353 359 367 373 379

383 389 397 401 409 419 421 431 433 439 443 449 457 461 463

467 479 487 491 499 503 509 521 523 541 547 557 563 569 571

577 587 593 599 601 607 613 617 619 631 641 643 647 653 659

661 673 677 683 691 701 709 719 727 733 739 743 751 757 761

769 773 787 797 809 811 821 823 827 829 839 853 857 859 863

877 881 883 887 907 911 919 929 937 941 947 953 967 971 977

983 991 997 1009 1013 1019 1021 1031 1033 1039 1049 1051 1061 1063 1069

1087 1091 1093 1097 1103 1109 1117 1123 1129 1151 1153 1163 1171 1181 1187

1193 1201 1213 1217 1223 1229 1231 1237 1249 1259 1277 1279 1283 1289 1291

1297 1301 1303 1307 1319 1321 1327 1361 1367 1373 1381 1399 1409 1423 1427

1429 1433 1439 1447 1451 1453 1459 1471 1481 1483 1487 1489 1493 1499 1511

1523 1531 1543 1549 1553 1559 1567 1571 1579 1583 1597 1601 1607 1609 1613

1619 1621 1627 1637 1657 1663 1667 1669 1693 1697 1699 1709 1721 1723 1733

1741 1747 1753 1759 1777 1783 1787 1789 1801 1811 1823 1831 1847 1861 1867

1871 1873 1877 1879 1889 1901 1907 1913 1931 1933 1949 1951 1973 1979 1987

1993 1997 1999 2003 2011 2017 2027 2029 2039 2053 2063 2069 2081 2083 2087

2089 2099 2111 2113 2129 2131 2137 2141 2143 2153 2161 2179 2203 2207 2213

2221 2237 2239 2243 2251 2267 2269 2273 2281 2287 2293 2297 2309 2311 2333

2339 2341 2347 2351 2357 2371 2377 2381 2383 2389 2393 2399 2411 2417 2423

2437 2441 2447 2459 2467 2473 2477 2503 2521 2531 2539 2543 2549 2551 2557

2579 2591 2593 2609 2617 2621 2633 2647 2657 2659 2663 2671 2677 2683 2687

2689 2693 2699 2707 2711 2713 2719 2729 2731 2741 2749 2753 2767 2777 2789

2791 2797 2801 2803 2819 2833 2837 2843 2851 2857 2861 2879 2887 2897 2903

2909 2917 2927 2939 2953 2957 2963 2969 2971 2999 3001 3011 3019 3023 3037

3041 3049 3061 3067 3079 3083 3089 3109 3119 3121 3137 3163 3167 3169 3181

3187 3191 3203 3209 3217 3221 3229 3251 3253 3257 3259 3271 3299 3301 3307

3313 3319 3323 3329 3331 3343 3347 3359 3361 3371 3373 3389 3391 3407 3413

3433 3449 3457 3461 3463 3467 3469 3491 3499 3511 3517 3527 3529 3533 3539

3541 3547 3557 3559 3571 3581 3583 3593 3607 3613 3617 3623 3631 3637 3643

3659 3671 3673 3677 3691 3697 3701 3709 3719 3727 3733 3739 3761 3767 3769

3779 3793 3797 3803 3821 3823 3833 3847 3851 3853 3863 3877 3881 3889 3907

3911 3917 3919 3923 3929 3931 3943 3947 3967 3989 4001 4003 4007 4013 4019

4021 4027 4049 4051 4057 4073 4079 4091 4093 4099 4111 4127 4129 4133 4139

4153 4157 4159 4177 4201 4211 4217 4219 4229 4231 4241 4243 4253 4259 4261

4271 4273 4283 4289 4297 4327 4337 4339 4349 4357 4363 4373 4391 4397 4409

4421 4423 4441 4447 4451 4457 4463 4481 4483 4493 4507 4513 4517 4519 4523

4547 4549 4561 4567 4583 4591 4597 4603 4621 4637 4639 4643 4649 4651 4657

4663 4673 4679 4691 4703 4721 4723 4729 4733 4751 4759 4783 4787 4789 4793

4799 4801 4813 4817 4831 4861 4871 4877 4889 4903 4909 4919 4931 4933 4937

4943 4951 4957 4967 4969 4973 4987 4993 4999 5003 5009 5011 5021 5023 5039

5051 5059 5077 5081 5087 5099 5101 5107 5113 5119 5147 5153 5167 5171 5179

5189 5197 5209 5227 5231 5233 5237 5261 5273 5279 5281 5297 5303 5309 5323

5333 5347 5351 5381 5387 5393 5399 5407 5413 5417 5419 5431 5437 5441 5443

5449 5471 5477 5479 5483 5501 5503 5507 5519 5521 5527 5531 5557 5563 5569

5573 5581 5591 5623 5639 5641 5647 5651 5653 5657 5659 5669 5683 5689 5693

5701 5711 5717 5737 5741 5743 5749 5779 5783 5791 5801 5807 5813 5821 5827

5839 5843 5849 5851 5857 5861 5867 5869 5879 5881 5897 5903 5923 5927 5939

5953 5981 5987 6007 6011 6029 6037 6043 6047 6053 6067 6073 6079 6089 6091

6101 6113 6121 6131 6133 6143 6151 6163 6173 6197 6199 6203 6211 6217 6221

6229 6247 6257 6263 6269 6271 6277 6287 6299 6301 6311 6317 6323 6329 6337

6343 6353 6359 6361 6367 6373 6379 6389 6397 6421 6427 6449 6451 6469 6473

6481 6491 6521 6529 6547 6551 6553 6563 6569 6571 6577 6581 6599 6607 6619

6637 6653 6659 6661 6673 6679 6689 6691 6701 6703 6709 6719 6733 6737 6761

6763 6779 6781 6791 6793 6803 6823 6827 6829 6833 6841 6857 6863 6869 6871

6883 6899 6907 6911 6917 6947 6949 6959 6961 6967 6971 6977 6983 6991 6997

7001 7013 7019 7027 7039 7043 7057 7069 7079 7103 7109 7121 7127 7129 7151

7159 7177 7187 7193 7207 7211 7213 7219 7229 7237 7243 7247 7253 7283 7297

7307 7309 7321 7331 7333 7349 7351 7369 7393 7411 7417 7433 7451 7457 7459

7477 7481 7487 7489 7499 7507 7517 7523 7529 7537 7541 7547 7549 7559 7561

7573 7577 7583 7589 7591 7603 7607 7621 7639 7643 7649 7669 7673 7681 7687

7691 7699 7703 7717 7723 7727 7741 7753 7757 7759 7789 7793 7817 7823 7829

7841 7853 7867 7873 7877 7879 7883 7901 7907 7919

## First 25 Catalan numbers

|  |  |
| --- | --- |
| 1 | 1 |
| 2 | 1 |
| 3 | 2 |
| 4 | 5 |
| 5 | 14 |
| 6 | 42 |
| 7 | 132 |
| 8 | 429 |
| 9 | 1430 |
| 10 | 4862 |
| 11 | 16796 |
| 12 | 58786 |
| 13 | 208012 |
| 14 | 742900 |
| 15 | 2674440 |
| 16 | 9694845 |
| 17 | 35357670 |
| 18 | 129644790 |
| 19 | 477638700 |
| 20 | 1767263190 |
| 21 | 6564120420 |
| 22 | 24466267020 |
| 23 | 91482563640 |
| 24 | 343059613650 |
| 25 | 1289904147324 |
| 26 | 4861946401452 |
| 27 | 18367353072152 |
| 28 | 69533550916004 |
| 29 | 263747951750360 |
| 30 | 1002242216651360 |
| 31 | 3814986502092300 |

## First 100 powers of 2

2^0 1

2^1 2

2^2 4

2^3 8

2^4 16

2^5 32

2^6 64

2^7 128

2^8 256

2^9 512

2^10 1024

2^11 2048

2^12 4096

2^13 8192

2^14 16384

2^15 32768

2^16 65536

2^17 131072

2^18 262144

2^19 524288

2^20 1048576

2^21 2097152

2^22 4194304

2^23 8388608

2^24 16777216

2^25 33554432

2^26 67108864

2^27 134217728

2^28 268435456

2^29 536870912

2^30 1073741824

2^31 2147483648

2^32 4294967296

2^33 8589934592

2^34 17179869184

2^35 34359738368

2^36 68719476736

2^37 137438953472

2^38 274877906944

2^39 549755813888

2^40 1099511627776

2^41 2199023255552

2^42 4398046511104

2^43 8796093022208

2^44 17592186044416

2^45 35184372088832

2^46 70368744177664

2^47 140737488355328

2^48 281474976710656

2^49 562949953421312

2^50 1125899906842624

2^51 2251799813685248

2^52 4503599627370496

2^53 9007199254740992

2^54 18014398509481984

2^55 36028797018963968

2^56 72057594037927936

2^57 144115188075855872

2^58 288230376151711744

2^59 576460752303423488

2^60 1152921504606846976

2^61 2305843009213693952

2^62 4611686018427387904

2^63 9223372036854775808

2^64 18446744073709551616

2^65 36893488147419103232

2^66 73786976294838206464

2^67 147573952589676412928

2^68 295147905179352825856

2^69 590295810358705651712

2^70 1180591620717411303424

2^71 2361183241434822606848

2^72 4722366482869645213696

2^73 9444732965739290427392

2^74 18889465931478580854784

2^75 37778931862957161709568

2^76 75557863725914323419136

2^77 151115727451828646838272

2^78 302231454903657293676544

2^79 604462909807314587353088

2^80 1208925819614629174706176

2^81 2417851639229258349412352

2^82 4835703278458516698824704

2^83 9671406556917033397649408

2^84 19342813113834066795298816

2^85 38685626227668133590597632

2^86 77371252455336267181195264

2^87 154742504910672534362390528

2^88 309485009821345068724781056

2^89 618970019642690137449562112

2^90 1237940039285380274899124224

2^91 2475880078570760549798248448

2^92 4951760157141521099596496896

2^93 9903520314283042199192993792

2^94 19807040628566084398385987584

2^95 39614081257132168796771975168

2^96 79228162514264337593543950336

2^97 158456325028528675187087900672

2^98 316912650057057350374175801344

2^99 633825300114114700748351602688

2^100 1267650600228229401496703205376

2^101 2535301200456458802993406410752

2^102 5070602400912917605986812821504

2^103 10141204801825835211973625643008

2^104 20282409603651670423947251286016

2^105 40564819207303340847894502572032

2^106 81129638414606681695789005144064

2^107 162259276829213363391578010288128

2^108 324518553658426726783156020576256

2^109 649037107316853453566312041152512

2^110 1298074214633706907132624082305024

2^111 2596148429267413814265248164610048

2^112 5192296858534827628530496329220096

2^113 10384593717069655257060992658440192

2^114 20769187434139310514121985316880384

2^115 41538374868278621028243970633760768

2^116 83076749736557242056487941267521536

2^117 166153499473114484112975882535043072

2^118 332306998946228968225951765070086144

2^119 664613997892457936451903530140172288

2^120 1329227995784915872903807060280344576

2^121 2658455991569831745807614120560689152

2^122 5316911983139663491615228241121378304

2^123 10633823966279326983230456482242756608

2^124 21267647932558653966460912964485513216

2^125 42535295865117307932921825928971026432

2^126 85070591730234615865843651857942052864

2^127 170141183460469231731687303715884105728

2^128 340282366920938463463374607431768211456

2^129 680564733841876926926749214863536422912

2^130 1361129467683753853853498429727072845824

2^131 2722258935367507707706996859454145691648

2^132 5444517870735015415413993718908291383296

2^133 10889035741470030830827987437816582766592

2^134 21778071482940061661655974875633165533184

2^135 43556142965880123323311949751266331066368

2^136 87112285931760246646623899502532662132736

2^137 174224571863520493293247799005065324265472

2^138 348449143727040986586495598010130648530944

2^139 696898287454081973172991196020261297061888

2^140 1393796574908163946345982392040522594123776

2^141 2787593149816327892691964784081045188247552

2^142 5575186299632655785383929568162090376495104

2^143 11150372599265311570767859136324180752990208

2^144 22300745198530623141535718272648361505980416

2^145 44601490397061246283071436545296723011960832

2^146 89202980794122492566142873090593446023921664

2^147 178405961588244985132285746181186892047843328

2^148 356811923176489970264571492362373784095686656

2^149 713623846352979940529142984724747568191373312

2^150 1427247692705959881058285969449495136382746624

## First 30 Rows of Pascal Triangle

1

1 1

1 2 1

1 3 3 1

1 4 6 4 1

1 5 10 10 5 1

1 6 15 20 15 6 1

1 7 21 35 35 21 7 1

1 8 28 56 70 56 28 8 1

1 9 36 84 126 126 84 36 9 1

1 10 45 120 210 252 210 120 45 10 1

1 11 55 165 330 462 462 330 165 55 11 1

1 12 66 220 495 792 924 792 495 220 66 12 1

1 13 78 286 715 1287 1716 1716 1287 715 286 78 13 1

1 14 91 364 1001 2002 3003 3432 3003 2002 1001 364 91 14 1

1 15 105 455 1365 3003 5005 6435 6435 5005 3003 1365 455 105 15 1

1 16 120 560 1820 4368 8008 11440 12870 11440 8008 4368 1820 560 120 16 1

1 17 136 680 2380 6188 12376 19448 24310 24310 19448 12376 6188 2380 680 136 17 1

1 18 153 816 3060 8568 18564 31824 43758 48620 43758 31824 18564 8568 3060 816 153 18 1

1 19 171 969 3876 11628 27132 50388 75582 92378 92378 75582 50388 27132 11628 3876 969 171 19 1

1 20 190 1140 4845 15504 38760 77520 125970 167960 184756 167960 125970 77520 38760 15504 4845 1140 190 20 1

1 21 210 1330 5985 20349 54264 116280 203490 293930 352716 352716 293930 203490 116280 54264 20349 5985 1330 210 21 1

1 22 231 1540 7315 26334 74613 170544 319770 497420 646646 705432 646646 497420 319770 170544 74613 26334 7315 1540 231 22 1

1 23 253 1771 8855 33649 100947 245157 490314 817190 1144066 1352078 1352078 1144066 817190 490314 245157 100947 33649 8855 1771 253 23 1

1 24 276 2024 10626 42504 134596 346104 735471 1307504 1961256 2496144 2704156 2496144 1961256 1307504 735471 346104 134596 42504 10626 2024 276 24 1

1 25 300 2300 12650 53130 177100 480700 1081575 2042975 3268760 4457400 5200300 5200300 4457400 3268760 2042975 1081575 480700 177100 53130 12650 2300 300 25 1

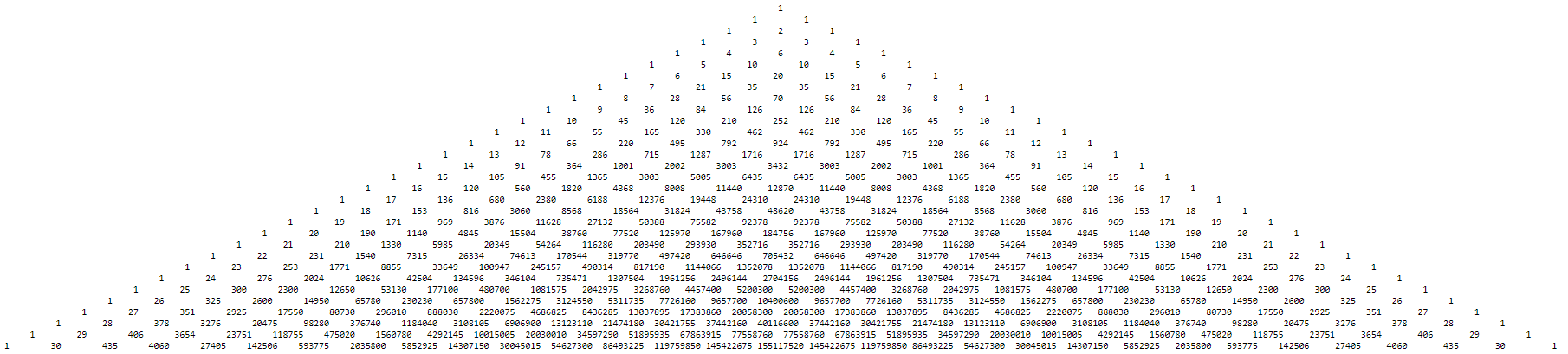
1 26 325 2600 14950 65780 230230 657800 1562275 3124550 5311735 7726160 9657700 10400600 9657700 7726160 5311735 3124550 1562275 657800 230230 65780 14950 2600 325 26 1

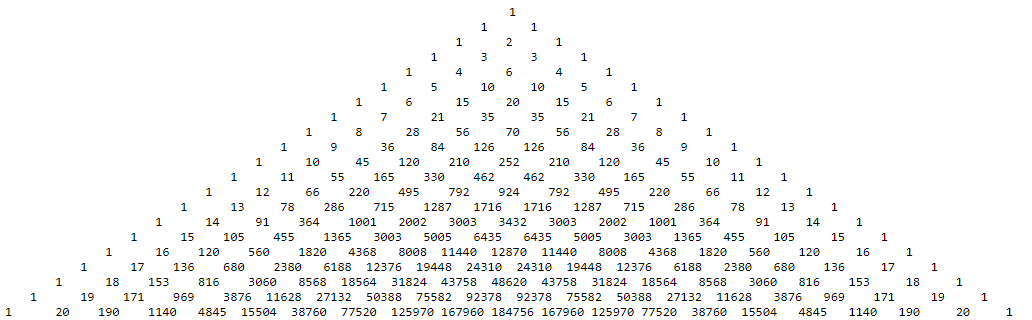
1 27 351 2925 17550 80730 296010 888030 2220075 4686825 8436285 13037895 17383860 20058300 20058300 17383860 13037895 8436285 4686825 2220075 888030 296010 80730 17550 2925 351 27 1

1 28 378 3276 20475 98280 376740 1184040 3108105 6906900 13123110 21474180 30421755 37442160 40116600 37442160 30421755 21474180 13123110 6906900 3108105 1184040 376740 98280 20475 3276 378 28 1

1 29 406 3654 23751 118755 475020 1560780 4292145 10015005 20030010 34597290 51895935 67863915 77558760 77558760 67863915 51895935 34597290 20030010 10015005 4292145 1560780 475020 118755 23751 3654 406 29 1

1 30 435 4060 27405 142506 593775 2035800 5852925 14307150 30045015 54627





## Código ASCII

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **ASCII** | **Hex** | **Symbol** | **ASCII** | **Hex** | **Symbol** | **ASCII** | **Hex** | **Symbol** | **ASCII** | **Hex** | **Symbol** |
| 0 | 0 | NUL | 32 | 20 | (space) | 64 | 40 | @ | 96 | 60 | ` |
| 1 | 1 | SOH | 33 | 21 | ! | 65 | 41 | A | 97 | 61 | a |
| 2 | 2 | STX | 34 | 22 | " | 66 | 42 | B | 98 | 62 | b |
| 3 | 3 | ETX | 35 | 23 | # | 67 | 43 | C | 99 | 63 | c |
| 4 | 4 | EOT | 36 | 24 | $ | 68 | 44 | D | 100 | 64 | d |
| 5 | 5 | ENQ | 37 | 25 | % | 69 | 45 | E | 101 | 65 | e |
| 6 | 6 | ACK | 38 | 26 | & | 70 | 46 | F | 102 | 66 | f |
| 7 | 7 | BEL | 39 | 27 | ' | 71 | 47 | G | 103 | 67 | g |
| 8 | 8 | BS | 40 | 28 | ( | 72 | 48 | H | 104 | 68 | h |
| 9 | 9 | TAB | 41 | 29 | ) | 73 | 49 | I | 105 | 69 | i |
| 10 | A | LF | 42 | 2A | \* | 74 | 4A | J | 106 | 6A | j |
| 11 | B | VT | 43 | 2B | + | 75 | 4B | K | 107 | 6B | k |
| 12 | C | FF | 44 | 2C | , | 76 | 4C | L | 108 | 6C | l |
| 13 | D | CR | 45 | 2D | - | 77 | 4D | M | 109 | 6D | m |
| 14 | E | SO | 46 | 2E | . | 78 | 4E | N | 110 | 6E | n |
| 15 | F | SI | 47 | 2F | / | 79 | 4F | O | 111 | 6F | o |
| 16 | 10 | DLE | 48 | 30 | 0 | 80 | 50 | P | 112 | 70 | p |
| 17 | 11 | DC1 | 49 | 31 | 1 | 81 | 51 | Q | 113 | 71 | q |
| 18 | 12 | DC2 | 50 | 32 | 2 | 82 | 52 | R | 114 | 72 | r |
| 19 | 13 | DC3 | 51 | 33 | 3 | 83 | 53 | S | 115 | 73 | s |
| 20 | 14 | DC4 | 52 | 34 | 4 | 84 | 54 | T | 116 | 74 | t |
| 21 | 15 | NAK | 53 | 35 | 5 | 85 | 55 | U | 117 | 75 | u |
| 22 | 16 | SYN | 54 | 36 | 6 | 86 | 56 | V | 118 | 76 | v |
| 23 | 17 | ETB | 55 | 37 | 7 | 87 | 57 | W | 119 | 77 | w |
| 24 | 18 | CAN | 56 | 38 | 8 | 88 | 58 | X | 120 | 78 | x |
| 25 | 19 | EM | 57 | 39 | 9 | 89 | 59 | Y | 121 | 79 | y |
| 26 | 1A | SUB | 58 | 3A | : | 90 | 5A | Z | 122 | 7A | z |
| 27 | 1B | ESC | 59 | 3B | ; | 91 | 5B | [ | 123 | 7B | { |
| 28 | 1C | FS | 60 | 3C | < | 92 | 5C | \ | 124 | 7C | | |
| 29 | 1D | GS | 61 | 3D | = | 93 | 5D | ] | 125 | 7D | } |
| 30 | 1E | RS | 62 | 3E | > | 94 | 5E | ^ | 126 | 7E | ~ |
| 31 | 1F | US | 63 | 3F | ? | 95 | 5F | \_ | 127 | 7F |  |

## Laws and facts

Sum of squares

An integer greater than one can be written as a sum of two squares if and only if its prime decomposition contains no prime congruent to 3 (mod 4) raised to an odd power.

Goldbach's conjecture

Every even integer greater than 2 can be expressed as the sum of two primes.

Is Divisible by prime

Given a number M, we want to know if it is divisible by a prime P.

1. Find the smallest K for this condition (KP + 1) MOD 10.
2. N = (KP + 1) / 10
3. Multiply the last digit of M times N and add the result to M without the last digit. (You might use (N-P) as well for the multiplication instead of N, note that it would be a negative number).
4. Repeat until M is short enough.