# Derangement:

**int derangement(int n)**

**{**

**if(!n) return n;**

**if(n <= 2) return n-1;**

**return (n-1)\*(derangement(n-1) + derangement(n-2));**

**}**

**/// Finds the number ways to put n balls into k indistinguishable boxes such that no box is empty`.**

**int stirling2(int n, int k)**

**{**

**if(n < k)**

**return 0;**

**if(k == 1)**

**return 1;**

**if(dp[n][k] == dp[n][k])**

**return dp[n][k];**

**return dp[n][k] = stirling2(n-1,k-1) + stirling2(n-1,k)\*k;**

**}**

**/// Finds the number of ways to put n elements into k cycles where no cycle is empty**

**int stirling1(int n, int k)**

**{**

**dp[n][k] = stirling1(n-1,k-1) + stirling(n-1,k)\*n-1;**

# }

# Sudoku Solver:

**char s[20];**

**int A[11][11], Row[11][11], Col[11][11], Box[11][11];**

**int boxNumber(int i, int j)**

**{**

**return ((i-1)/3 \* 3) + ceil(j/3.0);**

**}**

**int check(int i, int j)**

**{**

**bool candidate[11] = {0};**

**int b = boxNumber(i,j), k;**

**for(k = 1; k<=9; k++)**

**candidate[k] |= Row[i][k], candidate[k] |= Col[j][k], candidate[k] |= Box[b][k];**

**int ret = 0;**

**for(k = 1; k<=9; k++)**

**ret += (candidate[k] == 0);**

**return ret;**

**}**

**bool call()**

**{**

**int i, j, k;**

**pii nxt;**

**int mn = inf;**

**FRE(i,1,9)**

**{**

**FRE(j,1,9)**

**{**

**if(A[i][j] == 0)**

**{**

**int nw = check(i,j);**

**if(nw < mn)**

**{**

**mn = nw;**

**nxt = {i,j};**

**}**

**}**

**}**

**}**

**if(mn == 0)**

**return false;**

**if(mn == inf)**

**return true;**

**bool candidate[11] = {0};**

**i = nxt.xx, j = nxt.yy;**

**int b = boxNumber(i,j);**

**for(k = 1; k<=9; k++)**

**{**

**candidate[k] |= Row[i][k];**

**candidate[k] |= Col[j][k];**

**candidate[k] |= Box[b][k];**

**if(!candidate[k])**

**{**

**Row[i][k] = Col[j][k] = Box[b][k] = 1;**

**A[i][j] = k;**

**if(call())**

**return true;**

**Row[i][k] = Col[j][k] = Box[b][k] = 0;**

**A[i][j] = 0;**

**}**

**}**

**return false;**

**}**

**int main()**

**{**

**int i, j, k, cs, t;**

**sf(t);**

**FRE(cs,1,t)**

**{**

**mem(Row,0);**

**mem(Col,0);**

**mem(Box,0);**

**FRE(i,1,9)**

**{**

**scanf("%s",s);**

**for(j = 0; j<9; j++)**

**{**

**if(s[j] == '.')**

**A[i][j+1] = 0;**

**else**

**{**

**A[i][j+1] = s[j] - '0';**

**Row[i][s[j]-'0'] = 1;**

**Col[j+1][s[j]-'0'] = 1;**

**k = boxNumber(i,j+1);**

**Box[k][s[j]-'0'] = 1;**

**}**

**}**

**}**

**call();**

**pf("Case %d:\n",cs);**

**FRE(i,1,9)**

**{**

**FRE(j,1,9)**

**pf("%d",A[i][j]) ;**

**puts("");**

**}**

**}**

**return 0;**

**}**

# Rectangle Union:

**struct point{**

**LL x, y1, y2;**

**bool st;**

**}A[MAX\*2+10];**

**struct node{**

**LL on, val;**

**}tree[MAX\*32+10];**

**map<int,int> M;**

**vector<int> v;**

**LL ulta[MAX\*4+10];**

**void update(int nd, int st, int ed, int i, int j, int v)**

**{**

**if(st == ed)**

**return;**

**if(st > j || ed < i)**

**return;**

**int mid = (st+ed+1)/2, lft = 2\*nd, rght = lft+1;**

**if(i <= st && j >= ed)**

**{**

**tree[nd].val+=v;**

**if(tree[nd].val == 0)**

**tree[nd].on = tree[lft].on + tree[rght].on;**

**else**

**tree[nd].on = (ulta[ed]-ulta[st]);//getchar();**

**return;**

**}**

**if(ed -st == 1) return;**

**update(lft, st, mid, i, j, v);**

**update(rght, mid, ed, i, j, v);**

**if(!tree[nd].val)**

**tree[nd].on = tree[lft].on + tree[rght].on;**

**}**

**LL query()**

**{**

**return tree[1].on;**

**}**

**bool operator < (point a, point b)**

**{**

**return a.x < b.x;**

**}**

**int main()**

**{**

**int cs, t, i, j, k, n;**

**LL x1, y1, x2, y2;**

**sf(t);**

**FRE(cs,1,t)**

**{**

**mem(tree,0);**

**M.clear();**

**v.clear();**

**sf(n);**

**int mx = 1;**

**FRE(i,1,n)**

**{**

**sll(x1,y1);**

**sll(x2,y2);**

**v.pb(x1);**

**v.pb(x2);**

**v.pb(y1);**

**v.pb(y2);**

**A[i\*2-1].st = 1;**

**A[i\*2-1].x = x1;**

**A[i\*2-1].y1 = y1;**

**A[i\*2-1].y2 = y2;**

**A[i\*2].st = 0;**

**A[i\*2].x = x2;**

**A[i\*2].y1 = y1;**

**A[i\*2].y2 = y2;**

**}**

**sort(all(v));**

**un(v);**

**for(i = 0; i<v.sz; i++)**

**M[v[i]] = i+1, ulta[i+1] = v[i];**

**sort(A+1,A+2\*n+1);**

**LL nw = 0;**

**LL total = 0;**

**mx = v.sz;**

**for(i = 1; i<=2\*n; i++)**

**{**

**total += (A[i].x - nw) \* (query());**

**nw = A[i].x;**

**if(A[i].st == 1)**

**update(1,1,mx,M[A[i].y1],M[A[i].y2],1);**

**else**

**update(1,1,mx,M[A[i].y1],M[A[i].y2],-1);**

**}**

**pf("Case %d: %lld\n",cs,total);**

**}**

**return 0;**

**}**

# Number of solutions of Extended Euclid:

**/\***

**You have to find the number of solutions of the following equation:**

**Ax + By + C = 0**

**Where A, B, C, x, y are integers and x1 ≤ x ≤ x2 and y1 ≤ y ≤ y2.**

**\*/**

**typedef long long i64;**

**struct Euclid {**

**i64 x, y, d;**

**Euclid() {}**

**Euclid( i64 xx, i64 yy, i64 dd ) { x = xx, y = yy, d = dd; }**

**};**

**Euclid egcd( i64 a, i64 b ) {**

**if( !b ) return Euclid(1, 0, a);**

**Euclid r = egcd ( b, a % b );**

**return Euclid( r.y, r.x - a / b \* r.y, r.d );**

**}**

**i64 abs64( i64 a ) {**

**return a > 0 ? a : -a;**

**}**

**i64 myfloor( i64 a, i64 b ) {**

**i64 c = a / b;**

**if( (a % b) && a < 0 ) c--;**

**return c;**

**}**

**i64 myceil( i64 a, i64 b ) {**

**i64 c = a / b;**

**if( (a % b) && a > 0 ) c++;**

**return c;**

**}**

**i64 solve() {**

**i64 a, b, c, x1, x2, y1, y2, X, Y, n1, n2, m1, m2;**

**scanf("%lld %lld %lld %lld %lld %lld %lld", &a, &b, &c, &x1, &x2, &y1, &y2);**

**c \*= -1;**

**if( a < 0 ) a \*= -1, x1 \*= -1, x2 \*= -1, swap(x1, x2);**

**if( b < 0 ) b \*= -1, y1 \*= -1, y2 \*= -1, swap(y1, y2);**

**if( !a && !b ) return !c ? (x2 - x1 + 1) \* (y2 - y1 + 1) : 0;**

**if( b == 0 ) {**

**if( c % a ) return 0;**

**i64 x = c / a;**

**return (x >= x1 && x <= x2) \* (y2 - y1 + 1);**

**}**

**if( a == 0 ) {**

**if( c % b ) return 0;**

**i64 y = c / b;**

**return ( y >= y1 && y <= y2 ) \* (x2 - x1 + 1);**

**}**

**Euclid s;**

**s = egcd( a, b );**

**if( c % s.d ) return 0;**

**a /= s.d;**

**b /= s.d;**

**c /= s.d;**

**s.d = 1;**

**X = s.x \* c;**

**Y = s.y \* c;**

**n2 = min( myfloor( X - x1, b ), myfloor( y2 - Y, a ) );**

**n1 = -min( myfloor( Y - y1, a ), myfloor( x2 - X, b ) );**

**return (n2 < n1) ? 0 : n2 - n1 + 1;**

**}**

# Convex Hull Trick:

**struct line {**

**LL m, c;**

**double intersect(const line p) const {**

**LL a = p.c-c, b = m-p.m;**

**return (long double)a / (long double)b;**

**// LL g = \_\_gcd(a, b);**

**// return { a/g, b/g };**

**}**

**inline LL getY(LL x) {**

**return m\*x + c;**

**}**

**};**

**namespace cht {**

**line arr[MAX];**

**int key;**

**bool minFlag;**

**void init(bool f) {**

**key = 0;**

**minFlag = f;**

**}**

**inline bool check(line a, line b, line c) {**

**auto ac = a.intersect(c);**

**auto ab = a.intersect(b);**

**return (ac > ab);**

**// return (1.0 \* ac.uu \* ab.vv > 1.0 \* ac.vv \* ab.uu);**

**}**

**inline void add(line l) {**

**if(key == 0) arr[++key] = l;**

**else if(arr[key].m == l.m) {**

**if(minFlag) {**

**if(arr[key].c < l.c) return;**

**else arr[key] = l;**

**}**

**else {**

**if(arr[key].c > l.c) return;**

**else arr[key] = l;**

**}**

**}**

**else {**

**while(key >= 2) {**

**if(!check(arr[key-1], arr[key], l)) key--;**

**else break;**

**}**

**arr[++key] = l;**

**}**

**}**

**inline bool onSegment(int idx, LL x) {**

**if(idx != 1) {**

**auto p = arr[idx].intersect(arr[idx-1]);**

**if(p > x) return false;**

**// if(1.0 \* p.uu > 1.0 \* p.vv\*x) return false;**

**}**

**if(idx != key) {**

**auto p = arr[idx].intersect(arr[idx+1]);**

**if(p < x) return false;**

**// if(1.0 \* p.uu < 1.0 \* p.vv\*x) return false;**

**}**

**return true;**

**}**

**}**

**LL solve() {**

**LL ret = LLONG\_MAX;**

**cht::init(true);**

**cht::add({ height[n-1], 0 });**

**int last = cht::key;**

**for(int idx=n-2; idx>=0; idx--) {**

**// querying**

**last = min(last, cht::key);**

**while(last <= cht::key && !cht::onSegment(last, cost[idx])) last++;**

**ret = cht::arr[last].getY(cost[idx]);**

**// inserting**

**cht::add({ height[idx], ret });**

**}**

**return ret;**

**}**

# Divide & Conquer DP Optimization:

**int cost[MAX+10][MAX+10], G[MAX+10][MAX+10], cum[MAX+10][MAX+10], n, k;**

**int dp[MAX+10][MAX+10];**

**void pre()**

**{**

**// set the cost function**

**// Then set the base case**

**for(int i = 1; i<=n; i++)**

**dp[0][i] = 1e9;**

**}**

**void call(int group, int L, int R, int optL, int optR)**

**{**

**if(L > R)**

**return;**

**int mid = (L+R)/2;**

**int ret = INT\_MAX, idx;**

**int lim = min(optR, mid);**

**for(int i = optL; i<=lim; i++)**

**{**

**int cur = dp[group-1][i-1] + cost[i][mid];**

**if(cur <= ret)**

**{**

**ret = cur;**

**idx = i;**

**}**

**}**

**dp[group][mid] = ret;**

**call(group, L, mid-1, optL, idx);**

**call(group, mid+1, R, idx, optR);**

**}**

**int main()**

**{**

**int i, j, cs, t;**

**sff(n,k);**

**FRE(i,1,n)**

**{**

**FRE(j,1,n)**

**sf(G[i][j]);**

**}**

**pre();**

**for(int group = 1; group <= k; group++)**

**call(group, 1, n, 1, n);**

**printf("%d\n",dp[k][n]);**

**return 0;**

**}**

Catalan Number:

C_0 = 1 \quad \mbox{and} \quad C_{n+1}=\frac{2(2n+1)}{n+2}C_n,

Also

C_n = {2n\choose n} - {2n\choose n+1} = {1\over n+1}{2n\choose n} \quad\text{ for }n\ge 0,

The first few Catalan numbers for n = 1, 2, ... are 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796

* *Cn* is the number of **Dyck words**[[2]](https://en.wikipedia.org/wiki/Catalan_number" \l "cite_note-2) of length 2*n*. A Dyck word is a [string](https://en.wikipedia.org/wiki/String_(computer_science)) consisting of *n* X's and *n* Y's such that no initial segment of the string has more Y's than X's (see also [Dyck language](https://en.wikipedia.org/wiki/Dyck_language" \o "Dyck language)). For example, the following are the Dyck words of length 6:

XXXYYY     XYXXYY     XYXYXY     XXYYXY     XXYXYY.

* Re-interpreting the symbol X as an open [parenthesis](https://en.wikipedia.org/wiki/Bracket#Parentheses) and Y as a close parenthesis, *Cn* counts the number of expressions containing *n* pairs of parentheses which are correctly matched:

((()))     ()(())     ()()()     (())()     (()())

* *Cn* is the number of different ways *n* + 1 factors can be completely [parenthesized](https://en.wikipedia.org/wiki/Bracket) (or the number of ways of[associating](https://en.wikipedia.org/wiki/Associativity) *n* applications of a [binary operator](https://en.wikipedia.org/wiki/Binary_operator)). For *n* = 3, for example, we have the following five different parenthesizations of four factors:

((ab)c)d     (a(bc))d     (ab)(cd)     a((bc)d)     a(b(cd))

* *Cn* is the number of monotonic [lattice paths](https://en.wikipedia.org/wiki/Lattice_path) along the edges of a grid with *n* × *n* square cells, which do not pass above the diagonal. A monotonic path is one which starts in the lower left corner, finishes in the upper right corner, and consists entirely of edges pointing rightwards or upwards. Counting such paths is equivalent to counting Dyck words: X stands for "move right" and Y stands for "move up".

*Cn* is the number of different ways a [convex polygon](https://en.wikipedia.org/wiki/Convex_polygon) with *n* + 2 sides can be cut into [triangles](https://en.wikipedia.org/wiki/Triangle) by connecting vertices with [straight lines](https://en.wikipedia.org/wiki/Straight_line) (a form of [Polygon triangulation](https://en.wikipedia.org/wiki/Polygon_triangulation)).

* *Cn* is the number of rooted [binary trees](https://en.wikipedia.org/wiki/Binary_tree) with *n* internal nodes (*n* + 1 leaves). Illustrated in following Figure are the trees corresponding to *n* = 0,1,2 and 3. There are 1, 1, 2, and 5 respectively. Here, we consider as binary trees those in which each node has zero or two children, and the internal nodes are those that have children.
* Successive applications of a binary operator can be represented in terms of a full [binary tree](https://en.wikipedia.org/wiki/Binary_tree). (A rooted binary tree is *full* if every vertex has either two children or no children.) It follows that *Cn* is the number of full binary [trees](https://en.wikipedia.org/wiki/Tree_(graph_theory)) with *n* + 1 leaves:

Binary Bracketing:

Ok, to make it a little simpler to understand, pretend that the entire thing is enclosed in a set of brackets as well, e.g. (((xx)x)x) and ((xx)xx).  
  
Now, in binary bracketing, each set of brackets must contain exactly two things direcly. For example in (((xx)x)x), the inner set of brackets contains two x's, the middle set contains the inner set plus one x (two things in total), and the outer set contains the middle set and one x (two things in total again). But in ((xx)xx), the inner set contains two x's (ok so far), but the outer set contains the inner set plus *two* x's for a total of 3 things - so it can't be *binary* bracketed.

Total number of possible bracketing n+1 things = nth Catalan number

Super Catalan Numbers calculate all possible Bracketing

# Super Catalan Numbers

The super Catalan numbers are given by the [recurrence relation](http://mathworld.wolfram.com/RecurrenceRelation.html)

|  |
| --- |
| S(n)=(3(2n-3)S(n-1)-(n-3)S(n-2))/n |

he super Catalan numbers count the number of [lattice paths](http://mathworld.wolfram.com/LatticePath.html) with diagonal steps from (n,n) to (0,0) which do not touch the diagonal line x=y.

The first few super Catalan numbers are 1, 1, 3, 11, 45, 197,

# Extended Euclid + Number of solutions to the equation:

**/\***

**Find the number of integer solutions to the problem:**

**Ax + By = C**

**x1<=x<=x2 and y1<=y<=y2**

**Note:**

**1. Number of points with integer co-ordinates between (x,y) and (\_x,\_y) is:**

**gcd(dx,dy)+1 ( Including both the terminal points )**

**2. Next point of(\_x, \_y) with integer co-ordinates on the straight line Ax+By=C**

**(\_x+b/g, \_y-a/g).**

**\*/**

**struct triple{**

**LL x, y, g;**

**};**

**triple egcd(LL a, LL b)**

**{**

**if(b == 0)**

**return {1, 0, a};**

**triple tmp = egcd(b, a%b);**

**triple ret;**

**ret.x = tmp.y;**

**ret.y = (tmp.x - (a/b)\*tmp.y);**

**ret.g = tmp.g;**

**return ret;**

**}**

**LL myCeil(LL a, LL b)**

**{**

**LL ret = (a/b);**

**if((a >= 0 && b>= 0) || (a < 0 && b < 0))**

**{**

**if(a%b)**

**ret++;**

**}**

**return ret;**

**}**

**LL myFloor(LL a, LL b)**

**{**

**LL ret = (a/b);**

**if((a > 0 && b < 0) || (a < 0 && b > 0))**

**{**

**if(a%b)**

**ret--;**

**}**

**return ret;**

**}**

**/// x1 <= \_x + t\*(b/g) <= x2**

**pll get\_range(LL x1, LL x2, LL \_x, LL b, LL g)**

**{**

**if((b/g) < 0)**

**swap(x1, x2);**

**LL low = myCeil(((x1 - \_x) \* g), b);**

**LL high = myFloor(((x2 - \_x) \* g), b);**

**return {low, high};**

**}**

**/// ax + by = c**

**LL solve(LL a, LL b, LL c, LL x1, LL x2, LL y1, LL y2)**

**{**

**triple sltn = egcd(a,b);**

**LL ret;**

**if(a == 0 && b == 0)**

**{**

**if(c == 0) ret = (x2 - x1 + 1) \* (y2 - y1 + 1);**

**else ret = 0;**

**}**

**else if(a == 0)**

**{**

**if(c%b) ret = 0;**

**else**

**{**

**LL y = (c/b);**

**if(y1 <= y && y <= y2)**

**ret = (x2 - x1 + 1);**

**else**

**ret = 0;**

**}**

**}**

**else if(b == 0)**

**{**

**if(c%a) ret = 0;**

**else**

**{**

**LL x = (c/a);**

**if(x1 <= x && x <= x2)**

**ret = (y2 - y1 + 1);**

**else**

**ret = 0;**

**}**

**}**

**else**

**{**

**if(c%sltn.g) ret = 0;**

**else**

**{**

**sltn.x \*= (c/sltn.g);**

**sltn.y \*= (c/sltn.g);**

**pll rangeX = get\_range(x1,x2,sltn.x,b,sltn.g);**

**pll rangeY = get\_range(y1,y2,sltn.y,-a,sltn.g);**

**pll range;**

**range = {max(rangeX.xx, rangeY.xx), min(rangeX.yy, rangeY.yy)};**

**ret = range.yy - range.xx + 1;**

**ret = max(0LL, ret);**

**}**

**}**

**return ret;**

**}**