ACM Notebook

UNAM I

“Es un greddy trivial”

Índice

DP y Combinatoria

DP del scoreboard

En una competencia con n equipos y posibilidad de empates, ¿cuántos posibles scoreboards finales puede haber? O lo que es lo mismo, dados n objetos de diferentes tamaños (algunos posiblemente iguales), ¿cuántos posibles ordenamientos puede haber usando las relaciones “<” e “=”?

Simply put DP: let $P[n]$ denote the number of ways n horses could finish the race; all operations will be done mod m.

Out of n horses, any $1 \le k \le n$ could've taken first place; the number of ways to choose k horses is . The remaining $n - k$ horses finished in 2nd..x-th place, which is the same as a result of a race with $n - k$ horses. Therefore, we get a recurrence relation

with P[0]=1.

Estructura de Datos

Dynamic Convex Hull Trick (C++11)

```
const ll is_query = -(1LL<<62);
struct Line {
    ll m, b;
    mutable function<const Line*()> succ;
    bool operator<(const Line& rhs) const {
        if (rhs.b != is_query) return m < rhs.m;
        const Line* s = succ();
        if (!s) return 0;
        ll x = rhs.m;
        return b - s->b < (s->m - m) * x;     // cambiar para mínimo
    }
    Line(ll _m, ll _b){ m = _m; b = _b; }
};
struct HullDynamic : public multiset<Line> { // will maintain upper hull for maximum
```

```
    bool bad(iterator y) {
        auto z = next(y);
        if (y == begin()) {
            if (z == end()) return 0;
            return y->m == z->m && y->b <= z->b;  // cambiar para mínimo
        }
        auto x = prev(y);
        if (z == end()) return y->m == x->m && y->b <= x->b;  // cambiar para mínimo
        return (x->b - y->b)*(z->m - y->m) >= (y->b - z->b)*(y->m - x->m);  // cambiar
para mínimo

                  // puede causar overflow, cambiar por:
            // (x->b - y->b)/(double)(y->m - x->m) >= (y->b - z->b)/(double)(z->m - y-
>m)
    }
    void insert_line(ll m, ll b) {
        auto y = insert( Line(m, b) );
        y->succ = [=] { return next(y) == end() ? 0 : &*next(y); };
        if (bad(y)) { erase(y); return; }
        while (next(y) != end() && bad(next(y))) erase(next(y));
        while (y != begin() && bad(prev(y))) erase(prev(y));
    }
    ll eval(ll x) {
        auto l = *lower_bound(Line(x, is_query));
        return l.m * x + l.b;
    }
};
```

Segment Tree

Los intervalos son de la forma [l, r). Las hojas se encuentran consecutivamente a partir de la n-ésima posición.

```
int tree[2*n];
void build()  // build the tree
{
      for (int i = n - 1; i > 0; --i)
             tree[i] = tree[i<<1] + tree[i<<1|1];
}

void modify(int p, int value)    // set value at position p
{
      for (tree[p += n] = value; p > 1; p >>= 1)
            tree[p>>1] = tree[p] + tree[p^1];
}

int query(int l, int r)   // sum on interval [l, r)
{
```

```
    int res = 0;
    for (l += n, r += n; l < r; l >>= 1, r >>= 1)
    {
        if (l&1) res += tree[l++];
        if (r&1) res += tree[--r];
    }
    return res;
}
```

Segment Tree - Lazy Updates

Se consideran updates de la forma arr[a] = v, arr[a+1] = v, ..., arr[b] = v. Para cada query se obtiene min(arr[a], arr[a+1], ..., arr[b]). Declarar tree[4n], lazy[4n]

```
int izq(int x){ return (x << 1)+1; }
int der(int x){ return (x << 1)+2; }

void init(int i, int j, int nodo)
{
    lazy[nodo] = 0;
    if (i == j)
    {
        tree[nodo] = arr[i];
        return;
    }
    int m = (i + j) >> 1;
    init(i, m, izq(nodo));
    init(m+1, j, der(nodo));
    tree[nodo] = min(tree[izq(nodo)], tree[der(nodo)]);
}

void update(int a, int b, int v, int i, int j, int nodo)
{
    if (lazy[nodo])
    {
        tree[nodo] = lazy[nodo];
        if (i != j)
        {
            lazy[izq(nodo)] = lazy[nodo];
            lazy[der(nodo)] = lazy[nodo];
        }
        lazy[nodo] = 0;
    }
    if (i > j or i > b or j < a)
        return;
    if (i >= a and j <= b)
    {
        tree[nodo] = v;
        if (i != j)
        {
```

```
                lazy[izq(nodo)] = v;
                lazy[der(nodo)] = v;
            }
            return;
      }
      int m = (i + j) >> 1;
      update(a, b, v, i, m, izq(nodo));
      update(a, b, v, m+1, j, der(nodo));
      tree[nodo] = min(tree[izq(nodo)], tree[der(nodo)]);
}

int query(int a, int b, int i, int j, int nodo)
{
      if (lazy[nodo])
      {
            tree[nodo] = lazy[nodo];
            if (i != j)
            {
                  lazy[izq(nodo)] = lazy[nodo];
                  lazy[der(nodo)] = lazy[nodo];
            }
            lazy[nodo] = 0;
      }
      if (a == i and b == j)
            return tree[nodo];
      int m = (i + j) >> 1;
      if (b <= m)
            return query(a, b, i, m, izq(nodo));
      else if (a > m)
            return query(a, b, m+1, j, der(nodo));
      else
            return min(query(a, m, i, m, izq(nodo)), query(m+1, b, m+1, j, der(nodo)));
}
```

Heavy-Light Decomposition

```
vector <vector <ii> > AdjList;
int chainNo = 0, chainHead[MAXN+5], chainPos[MAXN+5], chainInd[MAXN+5],
chainSize[MAXN+5];
int subsize[MAXN+5], depth[MAXN+5], parent[MAXN+5], distParent[MAXN+5];
int Id[MAXN+5], Id_[MAXN+5], id = 0;

bool mycomp(int A, int B){ return depth[A] > depth[B]; }
void add_edge(int u, int v, int w)
{
      AdjList[u].push_back(MP(w, v));
      AdjList[v].push_back(MP(w, u));
}
void HLD(int root, int n)
{
```

```
  memset(chainHead, -1, sizeof chainHead);
memset(chainSize, 0, sizeof chainSize);
chainNo = 0;
id = 0;

  queue <ii> cola;
  cola.push(MP(root, -1));
  depth[root] = 0;
  parent[root] = -1;
  while (!cola.empty())
  {
        int u = cola.front().first;
        int p = cola.front().second;
        cola.pop();

        For(i, 0, (int)AdjList[u].size())
        {
              int v = AdjList[u][i].second;
              int w = AdjList[u][i].first;
              if (v != p)
              {
                    cola.push(MP(v, u));
                    depth[v] = depth[u]+1;
                    parent[v] = u;
                    distParent[v] = w;
              }
        }
  }

  int orden[n];
  For(i, 0, n)
        orden[i] = i;
  sort(orden, orden+n, mycomp);

  For(k, 0, n)
  {
        int u = orden[k];
        subsize[u] = 1;

        For(i, 0, (int)AdjList[u].size())
        {
              int v = AdjList[u][i].second;
              if (depth[v] > depth[u])
                    subsize[u] += subsize[v];
        }
  }

  cola.push(MP(root, chainNo));
  while (!cola.empty())
  {
```

```
        int u     = cola.front().first;
        int chain = cola.front().second;
        cola.pop();

        if (chain == -1)
              chain = ++chainNo;

        if (chainHead[chain] == -1)
              chainHead[chain] = u;

        chainInd[u] = chain;
        chainPos[u] = chainSize[chain];
        ++chainSize[chain];

        int sp = -1, maxi = -1;
        For(i, 0, (int)AdjList[u].size())
        {
              int v = AdjList[u][i].second;
              if (depth[v] < depth[u])
                    continue;

              if (subsize[v] > maxi)
                    sp = v, maxi = subsize[v];
        }

        if (sp != -1)
              cola.push(MP(sp, chain));

        For(i, 0, (int)AdjList[u].size())
        {
              int v = AdjList[u][i].second;
              if (v != sp and depth[v] > depth[u])
                    cola.push(MP(v, -1));
        }
  }

  For(k, 0, chainNo+1)
  {
        int root = chainHead[k];
        queue <int> q;
        q.push(root);
        Id[root] = id;
        Id_[id] = root;
        ++id;
        while (!q.empty())
        {
              int u = q.front(); q.pop();
              For(i, 0, (int)AdjList[u].size())
              {
                    int v = AdjList[u][i].second;
```

```
                    if (chainInd[v] == k and depth[v] > depth[u])
                        q.push(v), Id[v] = id, Id_[id] = v, ++id;
                }
            }
        }
}

/* Ejemplo query up para la trayectoria (a, b). r = LCA(a, b).
   Mandar llamar con x = a y x = b
   Estar atento a conteo doble sobre r */
int query_up(int x, int r)
{
        int ans = 0, u;
        for (u = x; chainInd[u] != chainInd[r]; u = parent[chainHead[chainInd[u]]])
            ans += query(Id[chainHead[chainInd[u]]], Id[u], 0, n-1, 0);
        ans += query(Id[r], Id[u], 0, n-1, 0);
        return ans;
}
```

Teoría de Grafos

König’s Theorem

En grafos bipartitos. |Minimum Vertex Cover| = |Maximum Matching|.
To construct such a cover, let U be the set of unmatched vertices in L (possibly empty), and let Z be the set of vertices that are either in U or are connected to U by alternating paths (paths that alternate between edges that are in the matching and edges that are not in the matching). Let $K = (L \setminus Z) \cup (R \cap Z)$.

Vertex Cover, Edge Cover & Independent Set

Un Vertex Cover es un conjunto de vértices tales que todas las aristas son incidentes a al menos un vértice en el conjunto.

Independent Set es un conjunto de vértices tales que para cualquier par de vértices en el conjunto no son adyacentes.

Ambos son problemas NP-Hard y son complementarios (si obtienes uno, puedes obtener el otro).

Edge Cover es un conjunto de aristas tales que cada vértice pertenece a alguna de las aristas. Es polinomial. Se obtiene encontrando el maximum matching y agregando aristas de forma greedy.

Número de árboles de expansión de un grafo (Kirchhoff’s theorem)

Matriz Laplaciana = Matriz de grados (matriz diagonal con grados de vértices en la diagonal) - Matriz de adyacencia.

Sean λ_1, λ_2, ..., λ_{n-1} los non-zero eigenvalores de la matriz Laplaciana. El número de árboles de expansión de G es:

$$t(G) = \frac{1}{n}\lambda_1\lambda_2\cdots\lambda_{n-1}\,.$$

Lowest Common Ancestor

```
Declarar int anc[log n][n], parent[n], depth[n]

int LCA(int a, int b)
{
    if (depth[a] < depth[b])
    {
        int t = a;
        a = b;
        b = t;
    }

    int i = 0;
    while (depth[a] > depth[b])
    {
        if ((depth[a]-depth[b]) & 1<<i)
            a = anc[i][a];
        i++;
    }

    if (a == b) return a;

    int s = 0;
    while (1 << (s+1) <= depth[a])
        s++;
    for (int i = s; i >= 0; i--)
        if (anc[i][a] != anc[i][b])
        {
            a = anc[i][a];
            b = anc[i][b];
        }

    return anc[0][a];
}
void init(int N)
{
    BFS(raiz) // Implementar para obtener depth[]
    For(i, 0, N)
      anc[0][i] = parent[i];
    for (int i = 0; 2<<i < N; i++)
      For(j, 0, N)
                anc[i+1][j] = anc[i][j] == -1 ? -1 : anc[i][anc[i][j]];
}
```

Flujo Máximo

```
Declarar vector<ii> path; vector <vector <edge> > AdjList; int f = 0 globales.
struct edge
{
	int v, rev, cap, flow;
	edge(int _v, int _rev, int _cap, int _flow)
	{
		v = _v;
		rev = _rev;
		cap = _cap;
		flow = _flow;
	}
	edge(){}
};

void addEdge(int u, int v, int cap)
{
	int k = AdjList[v].size(), l = AdjList[u].size();
	AdjList[u].push_back(edge(v, k, cap, 0));
	AdjList[v].push_back(edge(u, l, 0, 0));
}

void augment(int s, int v, int min_edge)
{
	if (v == s)
	{
		f = min_edge;
		return;
	}

	int u = path[v].first, i = path[v].second;
	if (u != -1)
	{
		int res = AdjList[u][i].cap - AdjList[u][i].flow;
		augment(s, u, min(res, min_edge));
		AdjList[u][i].flow += f;
		AdjList[v][AdjList[u][i].rev].flow -= f;
	}
}

int maxFlow(int s, int t, int N)
{
	int maxflow = 0;
	path.resize(N);
	while (true)
	{
		f = 0;
		vector <int> dist(N, INF);
		vector <bool> visit(N, false);
```

```
		path[t] = MP(-1, -1);


		queue <int> cola;
		cola.push(s);
		visit[s] = true;

		while (!cola.empty())
		{
			int u = cola.front(); cola.pop();
			if (u == t)
				break;

			For(i, 0, (int)AdjList[u].size())
			{
				edge e = AdjList[u][i];
				if (!visit[e.v] and e.cap - e.flow > 0)
					cola.push(e.v), visit[e.v] = true, path[e.v] = MP(u, i);
			}
		}

		augment(s, t, INF);
		if (!f)
			break;
		maxflow += f;
	}

	return maxflow;
}
```

Flujo Máximo de Costo Mínimo

Declarar `vector<ii> path; int f = 0, total_cost = 0; vector <vector <edge> > AdjList` globales.

```
struct edge
{
	int v, rev, cap, flow, cost;
	edge(int _v, int _rev, int _cap, int _flow, int _cost)
	{
		v = _v;
		rev = _rev;
		cap = _cap;
		flow = _flow;
		cost = _cost;
	}
	edge(){}
};
```

```
void addEdge(int u, int v, int cap, int cost)
{
      int k = AdjList[v].size(), l = AdjList[u].size();
      AdjList[u].push_back(edge(v, k, cap, 0, cost));
      AdjList[v].push_back(edge(u, l, 0, 0, -cost));
}

void augment(int s, int v, int min_edge)
{
      if (v == s)
      {
            f = min_edge;
            return;
      }
      int u = path[v].first, i = path[v].second;
      if (u != -1)
      {
            int res = AdjList[u][i].cap - AdjList[u][i].flow;
            augment(s, u, min(res, min_edge));
            AdjList[u][i].flow += f;
            AdjList[v][AdjList[u][i].rev].flow -= f;
            total_cost += AdjList[u][i].cost*f;
      }
}

int minCostMaxFlow(int s, int t, int N)
{
    int maxflow = 0;
    path.resize(N);
    while (true)
    {
        f = 0;
        vector <int> dist(N, INF);
        vector <bool> IN(N, false);
        dist[s] = 0;
        path[t] = MP(-1, -1);

        queue <int> cola;
        cola.push(s);
        IN[s] = true;

        while (!cola.empty())
        {
            int u = cola.front(); cola.pop();
            IN[u] = false;

            For(i, 0, (int)AdjList[u].size())
            {
                edge e = AdjList[u][i];
                if (e.cap - e.flow > 0 and dist[e.v] > dist[u] + e.cost)
```

```
                {
                    dist[e.v] = dist[u] + e.cost;
                    path[e.v] = MP(u, i);
                    if (!IN[e.v])
                        cola.push(e.v), IN[e.v] = true;
                }
            }
        }

        augment(s, t, INF);
        if (!f)
            break;
        maxflow += f;
    }

    return maxflow;
}
```

Matemáticas

Primitive Root

Si el orden multiplicativo de n módulo n es $\varphi(n)$, entonces m es raíz primitiva de n. Número de raíces primitivas de n es $\varphi(\varphi(n))$. Para saber si m es raíz primitva de n, obtener factores primos, p_i, de $\varphi(n)$ y calcular $m^{\varphi(n)/p_i}\ mod\ n$, si es diferente de 1 para cada p_i, entonces m es raíz primitiva.

Good primes

Primes less than 1000

// 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

Other primes

// The largest prime smaller than 10 is 7.
// The largest prime smaller than 100 is 97.
// The largest prime smaller than 1000 is 997.
// The largest prime smaller than 10000 is 9973.
// The largest prime smaller than 100000 is 99991.
// The largest prime smaller than 1000000 is 999983.
// The largest prime smaller than 10000000 is 9999991.
// The largest prime smaller than 100000000 is 99999989.
// The largest prime smaller than 1000000000 is 999999937.
// The largest prime smaller than 10000000000 is 9999999967.
// The largest prime smaller than 100000000000 is 99999999977.
// The largest prime smaller than 1000000000000 is 999999999989.
// The largest prime smaller than 10000000000000 is 9999999999971.
// The largest prime smaller than 100000000000000 is 99999999999973.
// The largest prime smaller than 1000000000000000 is 999999999999989.
// The largest prime smaller than 10000000000000000 is 9999999999999937.
// The largest prime smaller than 100000000000000000 is 99999999999999997.
// The largest prime smaller than 1000000000000000000 is 999999999999999989.

Gambler’s ruin problem

Dos jugadores tienen n1 y n2 monedas respectivamente. Lanzan volados y el ganador en cada ocasión toma una moneda del perdedor. Juegan hasta que uno de los dos se quede sin monedas. Asumiendo que los volados tienen resultados equiprobables, la probabilidad de que el jugador 1 se quede sin monedas al final del juego es:

$$P_1 = \frac{n_2}{n_1 + n_2}$$

Si las monedas de los volados están cargadas, de modo que el jugador 1 gana con probabilidad p y el jugador 2 gana con probabilidad q, la probabilidad de que el jugador 1 se quede sin monedas al final del juego será:

$$P_1 = \frac{1 - (\frac{p}{q})^{n_2}}{1 - (\frac{p}{q})^{n_1+n_2}}$$

Deragements

En combinatoria, un desordenamiento es una permutación de los elementos de un conjunto tal que ningún elemento aparece un su posición original. El número de desordenamientos de n elementos se denota por !n y está dado por la siguiente recurrencia:

O bien:

donde es la función entero más cercano y es la función piso.

Número de coloraciones diferentes para un cubo con n colores

El resultado es una aplicación del Lema de Burnside:

Lema de Burnside

Sea G un grupo que actúa en X. Para cada g en G, sea X^g el conjunto de elementos en X que están fijos por g. Entonces el número de órbitas es:

Problema de Josephus

```
Personas numeradas de 0 a n-1, empezando con la eliminación del k-ésimo.

long josephus(long n, long k){
 if(n==1) return 0;
 return (josephus(n-1,k)+k)%n;
}
```

Eurler’s theorem

$$\forall a, n \cdot gdc(a, n) = 1, a^{Phi(n)} \equiv 1 \ (mod\, N)$$

Fermat’s little theorem

Sea P un número primo.

$$\forall a \in \Re,\ a^P \equiv a\ (mod\ P)$$
$$\forall a \neq kP,\ a^{P-1} \equiv 1\ (mod\ P)$$

Divisor summatory function

$$\sum_{k=1}^{N} floor(N/k) = 2\sum_{k=1}^{u} floor(N/k) - u^2\ ,\ u = floor(sqrt(N))$$

Bayes' theorem

Sea P(A) la probabilidad de que suceda A y P(A|B), la probabilidad de que suceda A dado que ha sucedido B, entonces:

$$P(A\,|\,B) = \frac{P(A)P(B\,|\,A)}{P(B)}$$

Algoritmo Extendido de Euclides

```
Regresa (x, y) para la ecuación ax + by = gcd(a, b).

ii extended_euclid(int a, int b)
{
	if (b == 0)
		return MP(1, 0);
	ii t = extended_euclid(b, a % b);
	return MP(t.second, t.first - t.second*(a / b));
}
```

Now consider ax+by+cz=t. This is equivalent to solving gcd(a,b)w+cz=t, since any solution to the latter yields a solution of the former (by suitable choice of x and y so that ax+by=gcd(a,b)w)

Generación de ternas pitagóricas

Todas las ternas pitagóricas x^2 + y^2 = z^2 con x, y, z primos relativos tienen la forma:

$$(r^2 - s^2)^2 + (2rs)^2 = (r^2 + s^2)^2$$

Se pueden generar todas hasta n usando r,s < sqrt(n).

Recurrencias tipo Fibonacci

F(n) = 2 F(n−1) + 2 F(n−2)

Gives us the recurrence relation

$$r^n = 2\,(r^{n-1} + r^{n-2})$$

we divide by rn−2rn−2 to get

$$r^2 = 2\,(r+1) \Longrightarrow r^2 - 2\,r - 2 = 0$$

which is our characteristic equation. The characteristic roots are

$$\lambda 1 = 1 - \sqrt{3}$$

$$\lambda 2 = 1 + \sqrt{3}$$

Thus (because we have two different solutions)

$$F(n) = c1(1-\sqrt{3})^n + c2(1+\sqrt{3})^n$$

Where c1 and c2 are constants that are chosen based on the base cases.

Integración Numérica

```
#define EPS 1e-7
#define N 12

double R[N+1][N+1];

// a, b: Límities de integración
// F: apuntador de función a la función a integrar

double romberg(double a, double b, double (*F)(double))
{
      int i, j, k;
      double h = (b-a);

      R[0][0] = ( (*F)(a) + (*F)(b) ) * h / 2;

      for (i = 1; i <= N; ++i)
      {
            h = h / 2;
            double sum = 0;

            for (k = 1; k < (1 << i); k += 2)
                  sum += (*F)(a + k * h);
            R[i][0] = R[i-1][0] / 2 + sum * h;
            for (j = 1; j <= i; ++j)
                  R[i][j] = R[i][j-1] + (R[i][j-1] - R[i-1][j-1]) / ((1 << (2*j)) - 1);
      }

      return R[N][N];
```

```
}

double fc(double x) {
      return (sin(x) + x*x);
}

int main() {
      cout << romberg(0, 1000000, fc) << '\n';
}
```

Matrix Library

```
typedef long double LD;
LD EPS = 1e-8;
struct MATRIX
{
    int n,m;
    vector< vector<LD> > a;
    void resize(int x, int y, LD v=0.0)
    {
        n=x; m=y;
        a.resize(n);
        for(int i=0; i<n; i++) a[i].resize(m, v);
    }
    LD Gauss()
    // Row elimination based on the first n columns
    // if the first n columns is not invertible, kill yourself
    // otherwise, return the determinant of the first n columns
    {
        int i,j,k;
        LD det=1.0, r;
        for(i=0;i<n;i++)
        {
            for(j=i, k=-1; j<n; j++) if(fabs(a[j][i])>EPS)
            { k=j; j=n+1; }
            if(k<0) { n=0; return 0.0; }
            if(k != i) { swap(a[i], a[k]); det=-det; }
            r=a[i][i]; det*=r;
            for(j=i; j<m; j++) a[i][j]/=r;
            for(j=i+1; j<n; j++)
            {
                r=a[j][i];
                for(k=i; k<m; k++) a[j][k]-=a[i][k]*r;
            }
        }
        for(i=n-2; i>=0; i--)
        for(j=i+1; j<n; j++)
        {
```

```
            r=a[i][j];
            for(k=j; k<m; k++) a[i][k]-=r*a[j][k];
        }
        return det;
    }
    int inverse()
    // assume n=m. returns 0 if not invertible
    {
        int i, j, ii;
        MATRIX T; T.resize(n, 2*n);
        for(i=0;i<n;i++) for(j=0;j<n;j++) T.a[i][j]=a[i][j];
        for(i=0;i<n;i++) T.a[i][i+n]=1.0;
        T.Gauss();
        if(T.n==0) return 0;
        for(i=0;i<n;i++) for(j=0;j<n;j++) a[i][j]=T.a[i][j+n];
        return 1;
    }
    vector<LD> operator*(vector<LD> v)
    // assume v is of size m
    {
        vector<LD> rv(n, 0.0);
        int i,j;
        for(i=0;i<n;i++)
        for(j=0;j<m;j++)
        rv[i]+=a[i][j]*v[j];
        return rv;
    }
    MATRIX operator*(MATRIX M1)
    {
        MATRIX R;
        R.resize(n, M1.m);
        int i,j,k;
        for(i=0;i<n;i++)
        for(j=0;j<M1.m;j++)
        for(k=0;k<m;k++) R.a[i][j]+=a[i][k]*M1.a[k][j];
        return R;
    }
    void show()
    {
        int i,j;
        for(i=0;i<n;i++)
        {
            for(j=0;j<m;j++) printf("%15.10f ", (double)a[i][j]);
            printf("n");
        }
        printf("end of the show n");
    }
};
LD det(MATRIX &M)
// compute the determinant of M
```

```
{
    MATRIX M1=M;
    LD r=M1.Gauss();
    if(M1.n==0) return 0.0;
    return r;
}
vector<LD> solve(MATRIX& M, vector<LD> v)
// return the vector x such that Mx = v; x is empty if M is not invertible
{
    vector<LD> x;
    MATRIX M1=M;
    if(!M1.inverse()) return x;
    return M1*v;
}
void show(vector<LD> v)
{
    int i;
    for(i=0;i<v.size();i++) printf("%15.10f ", (double)v[i]);
    printf("n");
}
```

Números de Catalán

In combinatorial mathematics, the **Catalan numbers** form a sequence of natural numbers that occur in various counting problems, often involving recursively-defined objects. They are named after the Belgian mathematician Eugène Charles Catalan (1814–1894).

Using zero-based numbering, the *n*th Catalan number is given directly in terms of binomial coefficients by

$$C_n = \frac{1}{n+1}\binom{2n}{n} = \frac{(2n)!}{(n+1)!\,n!} = \prod_{k=2}^{n} \frac{n+k}{k} \qquad \text{for } n \geq 0.$$

The first Catalan numbers for $n = 0, 1, 2, 3, \ldots$ are

1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, 58786, 208012, 742900, 2674440, 9694845, 35357670, 129644790, 477638700, 1767263190, 6564120420, 24466267020, 91482563640, 343059613650, 1289904147324, 4861946401452

Properties

An alternative expression for C_n is

$$C_n = \binom{2n}{n} - \binom{2n}{n+1} = \frac{1}{n+1}\binom{2n}{n} \quad \text{for } n \geq 0,$$

which is equivalent to the expression given above because $\binom{2n}{n+1} = \frac{n}{n+1}\binom{2n}{n}$. This shows that C_n is an integer, which is not immediately obvious from the first formula given. This expression forms the basis for a proof of the correctness of the formula.

The Catalan numbers satisfy the recurrence relation

$$C_0 = 1 \quad \text{and} \quad C_{n+1}=\sum_{i=0}^{n}C_i\,C_{n-i}\quad\text{for }n\ge 0;$$

moreover,

$$C_n=\frac{1}{n+1}\sum_{i=0}^{n}\binom{n}{i}^2.$$

This is because $\binom{2n}{n} = \sum_{i=0}^{n} \binom{n}{i}^2,$ since choosing n numbers from a $2n$ set of numbers can be uniquely divided into 2 parts: choosing i numbers out of the first n numbers and then choosing n-i numbers from the remaining n numbers.

They also satisfy:

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

which can be a more efficient way to calculate them.

Asymptotically, the Catalan numbers grow as

$$C_n \sim \frac{4^n}{n^{3/2}\sqrt{\pi}}$$

in the sense that the quotient of the nth Catalan number and the expression on the right tends towards 1 as $n \to +\infty$. Some sources use just $C_n \sim \frac{4^n}{n^{3/2}}$.[1] (This can be proved by using Stirling's approximation for $n!$.)

The only Catalan numbers C_n that are odd are those for which $n = 2^k - 1$. All others are even.

The only prime Catalan numbers are $C_2 = 2$ and $C_3 = 5$.[citation needed]

The Catalan numbers have an integral representation

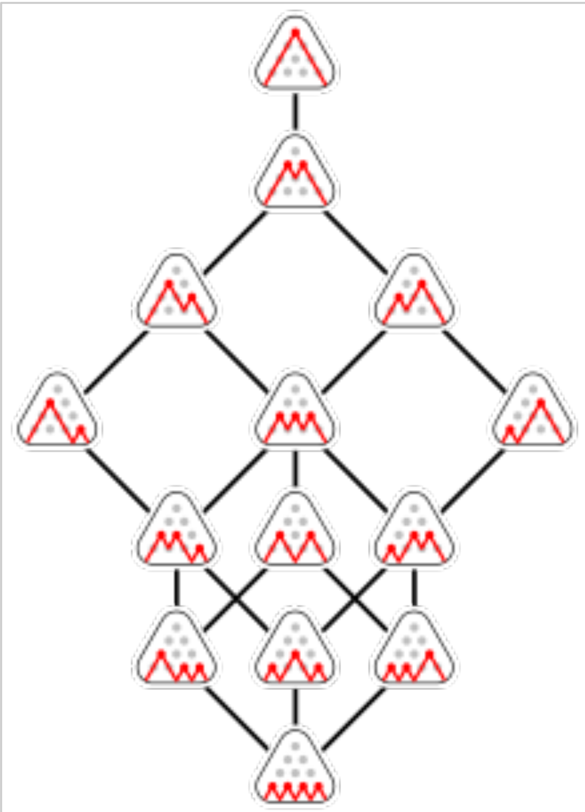
$$C_n=\int_0^4 x^n\rho(x)dx$$

where $\rho(x) = \frac{1}{2\pi}\sqrt{\frac{4-x}{x}}$. This means that the Catalan numbers are a solution of the Hausdorff moment problem on the interval [0, 4] instead of [0, 1]. The orthogonal polynomialshaving the weight function $\rho(x)$ on $[0,4]$ are

$$H_n(x) = \sum_{k=0}^{n} \binom{n+k}{n-k} (-x)^k.$$

Applications in combinatorics

There are many counting problems in combinatorics whose solution is given by the Catalan numbers. The book *Enumerative Combinatorics: Volume 2* by combinatorialist Richard P. Stanley contains a set of exercises which describe 66 different interpretations of the Catalan numbers. Following are some examples, with illustrations of the cases $C_3 = 5$ and $C_4 = 14$.



Lattice of the 14 Dyck words of length 8 - *(* and *)* interpreted as *up* and *down*

- C_n is the number of **Dyck words**[2] of length $2n$. A Dyck word is a string 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). For example, the following are the Dyck words of length 6:

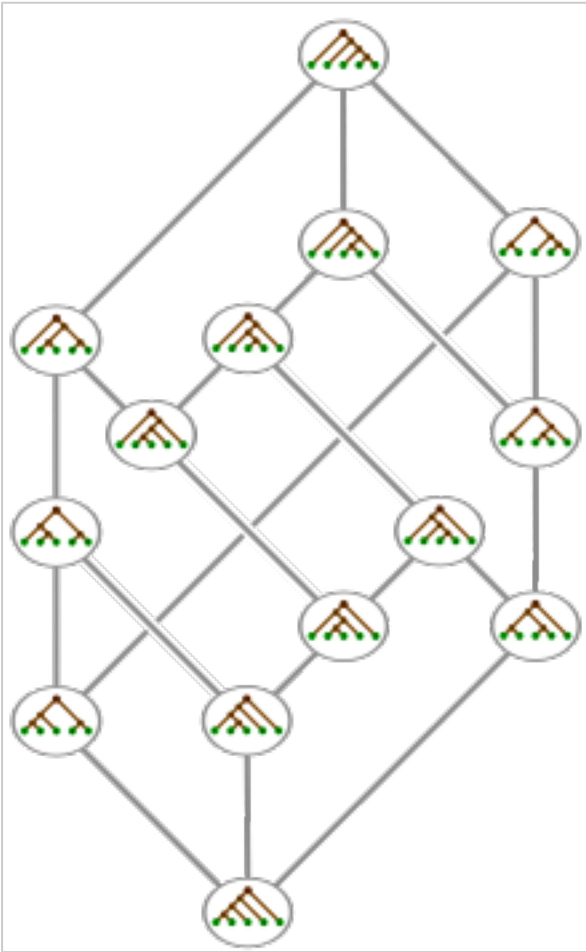
XXXYYY XYXXYY XYXYXY XXYYXY XXYXYY.

- Re-interpreting the symbol X as an open parenthesis and Y as a close parenthesis, C_n counts the number of expressions containing $n$pairs of parentheses which are correctly matched:

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

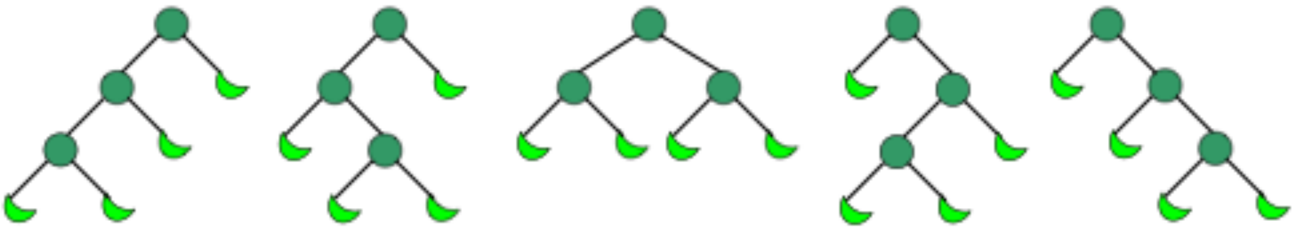
- C_n is the number of different ways n + 1 factors can be completely parenthesized (or the number of ways of associating n applications of a binary operator). For n = 3, for example, we have the following five different parenthesizations of four factors:

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



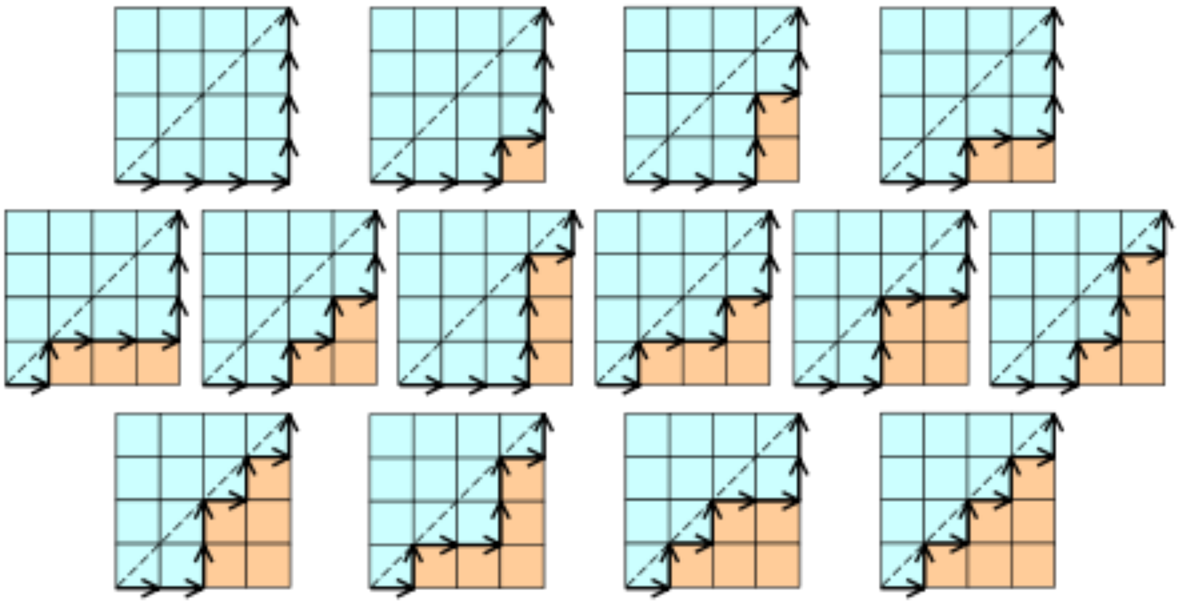
The associahedron of order 4 with the C_4=14 full binary trees with 5 leaves

- Successive applications of a binary operator can be represented in terms of a full binary tree. (A rooted binary tree is *full* if every vertex has either two children or no children.) It follows that C_n is the number of full binary trees with n + 1 leaves:



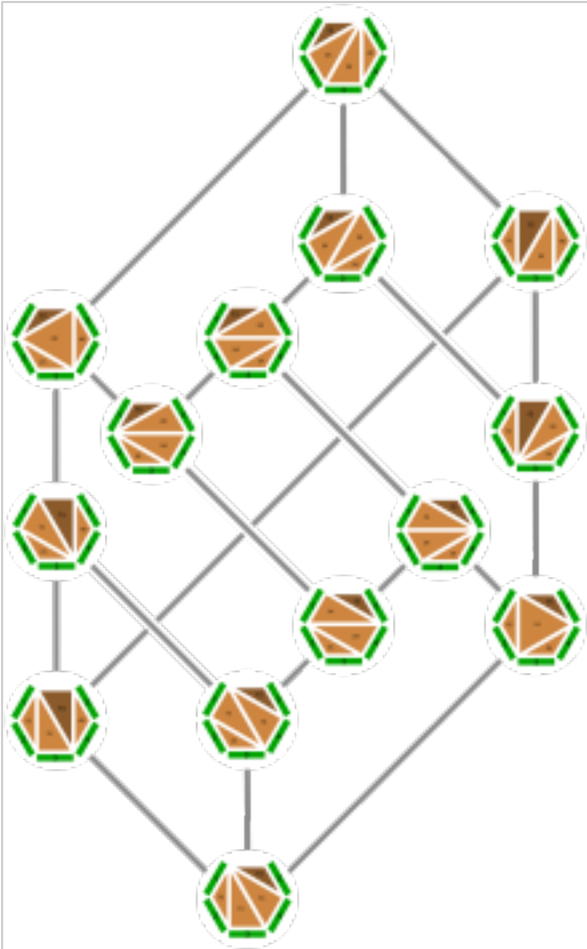
- C_n is the number of non-isomorphic ordered trees with n vertices. (An ordered tree is a rooted tree in which the children of each vertex are given a fixed left-to-right order.)[3]
- C_n is the number of monotonic lattice paths along the edges of a grid with $n \times n$ square cells, which do not 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".

The following diagrams show the case $n = 4$:



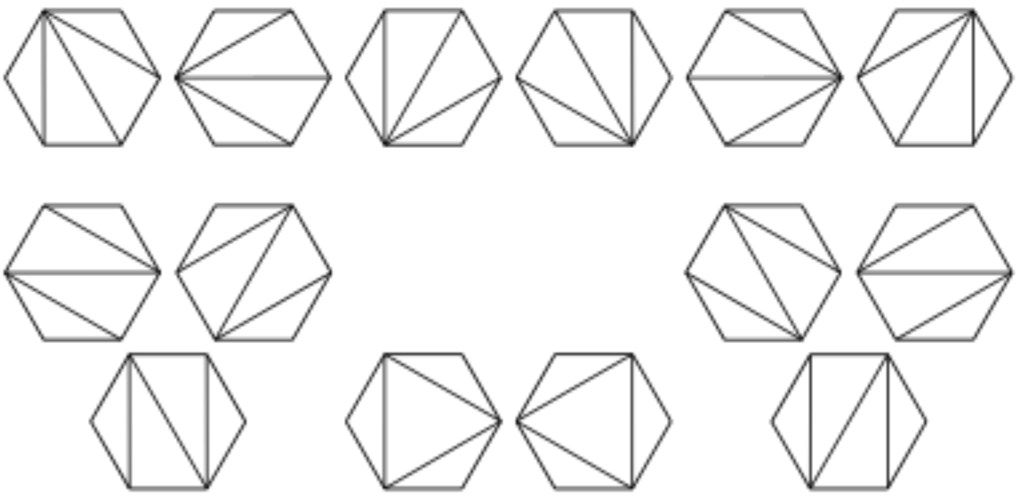
This can be succinctly represented by listing the Catalan elements by column height:[4]

[0,0,0,0][0,0,0,1][0,0,0,2][0,0,1,1]
[0,1,1,1] [0,0,1,2] [0,0,0,3] [0,1,1,2][0,0,2,2][0,0,1,3]
[0,0,2,3][0,1,1,3] [0,1,2,2][0,1,2,3]

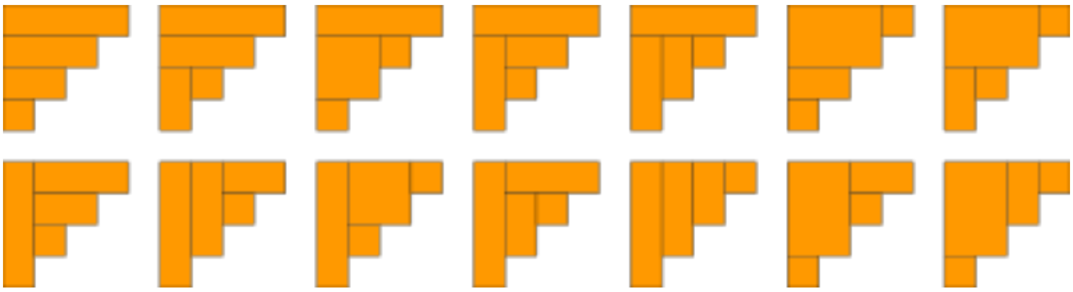


The triangles correspond to nodes of the binary trees.

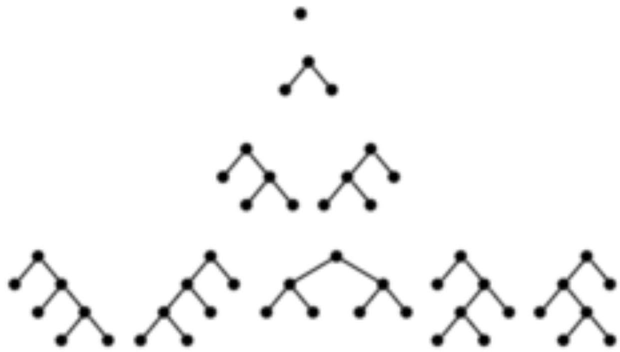
- C_n is the number of different ways a convex polygon with n + 2 sides can be cut into triangles by connecting vertices with straight lines (a form of Polygon triangulation). The following hexagons illustrate the case n = 4:



- C_n is the number of stack-sortable permutations of {1, ..., n}. A permutation w is called stack-sortable if $S(w) = (1, ..., n)$, where $S(w)$ is defined recursively as follows: write $w = unv$ where n is the largest element in w and u and v are shorter sequences, and set $S(w) = S(u)S(v)n$, with S being the identity for one-element sequences. These are the permutations that avoid the pattern 231.
- C_n is the number of permutations of {1, ..., n} that avoid the pattern 123 (or any of the other patterns of length 3); that is, the number of permutations with no three-term increasing subsequence. For $n = 3$, these permutations are 132, 213, 231, 312 and 321. For $n = 4$, they are 1432, 2143, 2413, 2431, 3142, 3214, 3241, 3412, 3421, 4132, 4213, 4231, 4312 and 4321.
- C_n is the number of noncrossing partitions of the set {1, ..., n}. *A fortiori*, C_n never exceeds the nth Bell number. C_n is also the number of noncrossing partitions of the set {1, ..., 2n} in which every block is of size 2. The conjunction of these two facts may be used in a proof bymathematical induction that all of the *free* cumulants of degree more than 2 of the Wigner semicircle law are zero. This law is important in free probability theory and the theory ofrandom matrices.
- C_n is the number of ways to tile a stairstep shape of height n with n rectangles. The following figure illustrates the case $n = 4$:



- C_n is the number of rooted binary trees with n internal nodes (n + 1 leaves or external nodes). 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.



Binary Trees

- C_n is the number of ways to form a “mountain ranges” with n upstrokes and n down-strokes that all stay above the original line.The mountain range interpretation is that the mountains will never go below the horizon.

$n = 0$:	*	1 way
$n = 1$:	/\	1 way
$n = 2$:	/\ /\/\, / \	2 ways
$n = 3$:	/\ /\ /\ /\/\ / \ /\/\/\, /\/ \, / \/\, / \, / \	5 ways

Mountain Ranges

- C_n is the number of standard Young tableaux whose diagram is a 2-by-n rectangle. In other words, it is the number of ways the numbers 1, 2, ..., $2n$ can be arranged in a 2-by-$n$rectangle so that each row and each column is increasing. As such, the formula can be derived as a special case of the hook-length formula.
- C_n is the number of ways that the vertices of a convex $2n$-gon can be paired so that the line segments joining paired vertices do not intersect. This is precisely the condition that guarantees that the paired edges can be identified (sewn together) to form a closed surface of genus zero (a topological 2-sphere).
- C_n is the number of semiorders on n unlabeled items.[5]
-

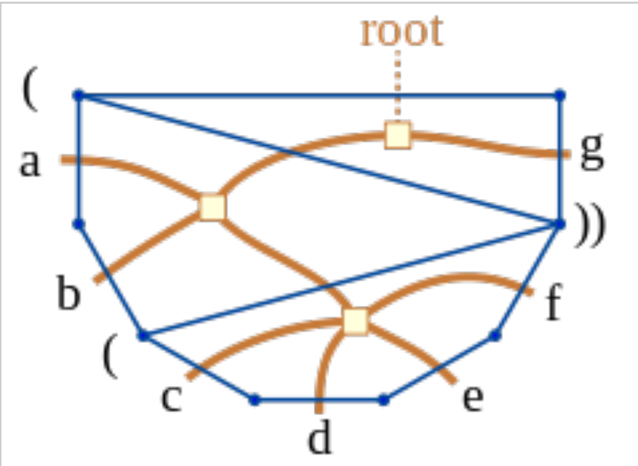
Súper Números de Catalán

In number theory, the **Schröder–Hipparchus numbers** form an integer sequence that can be used to count the number of plane trees with a given set of leaves, the number of ways of inserting parentheses into a sequence, and the number of ways of dissecting a convex polygon into smaller polygons by inserting diagonals. These numbers begin

1, 1, 3, 11, 45, 197, 903, 4279, 20793, 103049, ... (sequence A001003 in OEIS).

They are also called the **super-Catalan numbers**, the **little Schröder numbers**, or the **Hipparchus numbers**, after Eugène Charles Catalan and his Catalan numbers, Ernst Schröder and the closely related Schröder numbers, and the ancient Greek mathematician Hipparchus who appears from evidence in Plutarch to have known of these numbers.

Combinatorial enumeration applications



Combinatorial equivalence between subdivisions of a polygon, plane trees, and parenthesizations

The Schröder–Hipparchus numbers may be used to count several closely related combinatorial objects:[1][2][3][4]

The nth number in the sequence counts the number of different ways of subdividing of a polygon with $n + 1$ sides into smaller polygons by adding diagonals of the original polygon.

- The nth number counts the number of different plane trees with n leaves and with all internal vertices having two or more children.
- The nth number counts the number of different ways of inserting parentheses into a sequence of n symbols, with each pair of parentheses surrounding two or more symbols or parenthesized groups, and without any parentheses surrounding the entire sequence.
- The nth number counts the number of faces of all dimensions of an associahedron K_{n+1} of dimension $n - 1$, including the associahedron itself as a face, but not including the empty set. For instance, the

two-dimensional associahedron K_4 is a pentagon; it has five vertices, five faces, and one whole associahedron, for a total of 11 faces.

As the figure shows, there is a simple combinatorial equivalence between these objects: a polygon subdivision has a plane tree as a form of its dual graph, the leaves of the tree correspond to the symbols in a parenthesized sequence, and the internal nodes of the tree other than the root correspond to parenthesized groups. The parenthesized sequence itself may be written around the perimeter of the polygon with its symbols on the sides of the polygon and with parentheses at the endpoints of the selected diagonals. This equivalence provides a bijective proof that all of these kinds of objects are counted by a single integer sequence.[2]

The same numbers also count the number of double permutations (sequences of the numbers from 1 to n, each number appearing twice, with the first occurrences of each number in sorted order) that avoid the permutation patterns 12312 and 121323.[5]

Related sequences

The closely related large Schröder numbers are equal to twice the Schröder–Hipparchus numbers, and may also be used to count several types of combinatorial objects including certain kinds of lattice paths, partitions of a rectangle into smaller rectangles by recursive slicing, and parenthesizations in which a pair of parentheses surrounding the whole sequence of elements is also allowed. The Catalan numbers also count closely related sets of objects including subdivisions of a polygon into triangles, plane trees in which all internal nodes have exactly two children, and parenthesizations in which each pair of parentheses surrounds exactly two symbols or parenthesized groups.[3]

The sequence of Catalan numbers and the sequence of Schröder–Hipparchus numbers, viewed as infinite-dimensional vectors, are the unique eigenvectors for the first two in a sequence of naturally defined linear operators on number sequences.[6][7] More generally, the kth sequence in this sequence of integer sequences is $(x_1, x_2, x_3, ...)$ where the numbers x_n are calculated as the sums of Narayana numbers multiplied by powers of k:

$$x_n = \sum_{i=1}^{n} N(n,i)\, k^{i-1} = \sum_{i=1}^{n} \frac{1}{n} \binom{n}{i} \binom{n}{i-1} k^{i-1}.$$

Substituting $k = 1$ into this formula gives the Catalan numbers and substituting $k = 2$ into this formula gives the Schröder–Hipparchus numbers.[7]

In connection with the property of Schröder–Hipparchus numbers of counting faces of an associahedron, the number of vertices of the associahedron are given by the Catalan numbers. The corresponding numbers for the permutohedron are respectively the ordered Bell numbers and the factorials.

Recurrence

As well as the summation formula above, the Schröder–Hipparchus numbers may be defined by a recurrence relation:

$$S(n) = \frac{1}{n}\left((6n-9)S(n-1) - (n-3)S(n-2)\right).$$

Stanley proves this fact using generating functions[8] while Foata and Zeilberger provide a direct combinatorial proof.[9]

Cadenas

Manachers' Algorithm

Para encontrar la subcadena palindrómica más grande. En P[i] se guarda el tamaño del palíndromo más grande centrado en i. s es la cadena original y t la cadena modificada.

```
int longestPalindrome()
{
      int size = 2;
      t[0] = '$', t[1] = '#';
      For(i, 0, n)
      {
            t[size++] = s[i];
            t[size++] = '#';
      }
      t[size++] = '^';

      int C = 0, R = 0;
      For(i, 1, size-1)
      {
            int i_ = 2*C - i;
            P[i] = (R > i) ? min(R - i, P[i_]) : 0;

            while (t[i+1+P[i]] == t[i-1-P[i]])
                  P[i]++;

            if (i + P[i] > R)
            {
                  C = i;
                  R = i + P[i];
            }
      }

    int ans = 0;
      For(i, 1, size-1)
            ans = max(ans, P[i]);
```

```
      return ans;
}
```

Geometría

Área de un polígono

Vértices ordenados, v[n]=v[0]. El polígono puede o no ser convexo. No funciona para polígonos que se autointersecan.

$$A = \frac{1}{2} \sum_{i=0}^{n-1} (x_i \; y_{i+1} - x_{i+1} \; y_i)$$

```
struct vect{
 long double x,y;
 vect(long double a=0.0, long double b=0.0):x(a),y(b){}
};

long double Area(vect h[], long o){
 long double Ar=0.0;
 for(int ii=0;ii<o;ii++) Ar+=(h[ii].x*h[ii+1].y-h[ii+1].x*h[ii].y);
 if(Ar<0.0) Ar=-Ar;
 return Ar/2.0;
}
```

Centroide de un polígono

Vértices ordenados, v[n]=v[0]. El polígono puede o no ser convexo. No funciona para polígonos que se autointersecan. Para triángulos basta con promediar las coordenadas de los vértices.

$$C_{\mathrm{x}} = \frac{1}{6A} \sum_{i=0}^{n-1} (x_i + x_{i+1})(x_i \; y_{i+1} - x_{i+1} \; y_i)$$

$$C_{\mathrm{y}} = \frac{1}{6A} \sum_{i=0}^{n-1} (y_i + y_{i+1})(x_i \; y_{i+1} - x_{i+1} \; y_i)$$

```
struct vect{
 long double x,y;
 vect(long double a=0.0, long double b=0.0):x(a),y(b){}
 vect operator-(vect a){ return vect(x-a.x,y-a.y); }
 double operator/(vect a){ return x*a.y-y*a.x; }
};

vect Mass_Center(vect z[], long w){
 long double area=Area(z,w);
 vect ee=vect();
```

```
 for(int ii=0;ii<w;ii++) ee.x+=(z[ii].x+z[ii+1].x)*(z[ii].x*z[ii+1].y-
z[ii+1].x*z[ii].y);
 for(int ii=0;ii<w;ii++) ee.y+=(z[ii].y+z[ii+1].y)*(z[ii].x*z[ii+1].y-
z[ii+1].x*z[ii].y);
 ee.x/=6.0*area;
 ee.y/=6.0*area;
 return ee;
}
```

Great-Circle Distance

Dadas las latitudes y longitudes de dos puntos sobre una esfera hallar la longitud de la geodésica que los une. La siguiente implementación (CP3) emplea una fórmula con grandes errores de precisión para puntos cercanos (ángulos pequeños):

```
long double gcDistance(long double pLat, long double pLong, long double qLat, long
double qLong, long double radius){
 pLat*=PI/180.0; pLong*=PI/180.0;
 qLat*=PI/180.0; qLong*=PI/180.0;
 return radius*acos(cos(pLat)*cos(pLong)*cos(qLat)*cos(qLong)
+cos(pLat)*sin(pLong)*cos(qLat)*sin(qLong)+sin(pLat)*sin(qLat));
}
```

A continuación se presenta una implementación que resuelve los problemas para ángulos pequeños a cambio de problemas con ángulos grandes (máximo error en puntos antipodales), empleando la fórmula de Haversine:

$$d = 2r\arcsin\left(\sqrt{\operatorname{haversin}(\phi_2-\phi_1)+\cos(\phi_1)\cos(\phi_2)\operatorname{haversin}(\lambda_2-\lambda_1)}\right)$$
$$= 2r\arcsin\left(\sqrt{\sin^2\left(\frac{\phi_2-\phi_1}{2}\right)+\cos(\phi_1)\cos(\phi_2)\sin^2\left(\frac{\lambda_2-\lambda_1}{2}\right)}\right)$$

```
long double gcdHaversine(long double pLat, long double pLong, long double qLat, long
double qLong, long double radius){
 pLat*=PI/180.0; pLong*=PI/180.0;
 qLat*=PI/180.0; qLong*=PI/180.0;
 return radius*2.0*asin( sqrt(sin((pLat-qLat)/2.0)*sin((pLat-qLat)/
2.0)+cos(pLat)*cos(qLat)*sin((pLong-qLong)/2.0)*sin((pLong-qLong)/2.0)) );
}
```

Closest Pair

```
struct punto
{
      double x, y;
};
```

```
bool compareX(punto A, punto B)
{
	if (A.x != B.x)
		return A.x < B.x;
	return A.y < B.y;
}

bool compareY(punto A, punto B)
{
	if (A.y != B.y)
		return A.y < B.y;
	return A.x < B.x;
}

double distE(punto A, punto B)
{
	return hypot(A.x - B.x, A.y - B.y);
}

double distClosestPair(vector <punto> P, vector <punto> Q)
{
	if (P.size() == 2)
		return distE(*P.begin(), *(P.end()-1));
	if (P.size() == 1)
		return INF*INF+1;
	int m = P.size()>>1;
	vector <punto> Rx, Ry, Lx, Ly;
	Lx.assign(P.begin(), P.begin()+m);
	Rx.assign(P.begin()+m, P.end());
	int j = 0;
	For(i, 0, Q.size())
		if (Q[i].x < Lx[Lx.size()-1].x)
			Ly.push_back(Q[i]);
		else if (Q[i].x > Lx[Lx.size()-1].x)
			Ry.push_back(Q[i]);
		else
		{
			if (Q[i].y <= Lx[Lx.size()-1].y)
				Ly.push_back(Q[i]);
			else
				Ry.push_back(Q[i]);
		}
	double p = distClosestPair(Lx, Ly);
	double q = distClosestPair(Rx, Ry);
	double minDist = min(p, q), delta = sqrt(minDist);
	vector <punto> S;
	int Xm = (*(Lx.end()-1)).x;
	For(i, 0, Q.size())
		if (Q[i].x >= Xm-delta and Q[i].x <= Xm+delta)
```

```
                S.push_back(Q[i]);
        if (!S.size()) return minDist;
        For(i, 0, S.size()-1)
                for(int j=i+1, k=0; k<min(7, (int)(S.size()-(i+1))); j++, k++)
                {
                        double dist = distE(S[i], S[j]);
                        if (dist < minDist)
                                minDist = dist;
                }

        return minDist;
}

int main()
{
        sort(P.begin(), P.end(), compareX);
        sort(Q.begin(), Q.end(), compareY);
        double dist = distClosestPair(P, Q);
}
```

Java

```
import java.io.*;
import java.util.*;
import java.math.*;

public class Main{
        public static MyScanner sc = new MyScanner();
        public static PrintWriter out;


        public static void main(String[] args) {

                out = new PrintWriter(new BufferedOutputStream(System.out));

                // <!--inicio solucion-->

                // </!--fin solucion-->

                out.close();
        }

        public static class MyScanner {
                BufferedReader br;
                StringTokenizer st;

                public MyScanner() {
                        br = new BufferedReader(new InputStreamReader(System.in));
                }
```

```java
String next() {
      while (st == null || !st.hasMoreElements()) {
            try {
                  st = new StringTokenizer(br.readLine());
            }
            catch (IOException e) {
                  e.printStackTrace();
            }
      }
      return st.nextToken();
}

int nextInt() {
      return Integer.parseInt(next());
}

long nextLong() {
      return Long.parseLong(next());
}

double nextDouble() {
      return Double.parseDouble(next());
}

BigInteger nextBigInteger() {
      return new BigInteger(next());
}

BigInteger nextBigInteger(int radix) {
      return new BigInteger(next(), radix);
}

String nextLine() {

      String str = "";

      try {
            str = br.readLine();
      }
      catch (IOException e) {
            e.printStackTrace();
      }
      return str;
}

public boolean hasNext() {
      while (st == null || !st.hasMoreTokens()) {
            try {
                  String line = br.readLine();
                  if (line == null) {
```

```
                        return false;
                    }
                    st = new StringTokenizer(line);
                } catch (IOException e) {
                    throw new RuntimeException(e);
                }
            }
            return true;
        }

    }
}
```

Garden Fence

```
// Accepted, 7.772s

using namespace std;
#include <algorithm>
#include <iostream>
#include <iterator>
#include <numeric>
#include <sstream>
#include <fstream>
#include <cassert>
#include <climits>
#include <cstdlib>
#include <cstring>
#include <string>
#include <cstdio>
#include <vector>
#include <cmath>
#include <queue>
#include <deque>
#include <stack>
#include <list>
#include <map>
#include <set>

////////////// Prewritten code follows. Look down for
solution. ////////////////
#define foreach(x, v) for (typeof (v).begin() x=(v).begin(); x !=(v).end(); +
+x)
#define For(i, a, b) for (int i=(a); i<(b); ++i)
```

```cpp
#define D(x) cout << #x " is " << (x) << endl


const double EPS = 1e-9;

int cmp(double x, double y = 0, double tol = EPS) {

   return (x <= y + tol) ? (x + tol < y) ? -1 : 0 : 1;

}

///////////////////////// Solution starts
below. //////////////////////////////


const int MAXN = 2005;


struct Tree {

   int x, y, value, color;

};


Tree trees[MAXN], v[MAXN];


inline bool above(const Tree &t) {

   assert(t.x != 0 or t.y != 0);

   if (t.y == 0) return t.x > 0;

   return t.y > 0;

}


inline bool below(const Tree &t) {

   return !above(t);

}


bool compare(const Tree &a, const Tree &b) {
```

```
    long long cross = 1LL * a.x * b.y - 1LL * a.y * b.x;
    if (!above(a)) cross = -cross;
    if (!above(b)) cross = -cross;
    return cross > 0;
}


bool equal(const Tree &a, const Tree &b) {
    long long cross = 1LL * a.x * b.y - 1LL * a.y * b.x;
    return cross == 0;
}


int sweep(int n, int colorAbove) {
    int colorBelow = 1 - colorAbove;

    sort(v, v + n, compare);


    int score = 0;
    for (int i = 0; i < n; ++i) {
        if (above(v[i]) and v[i].color != colorAbove) score += v[i].value;
        if (below(v[i]) and v[i].color != colorBelow) score += v[i].value;
    }

    int ans = score;


    for (int i = 0, j; i < n; i = j) {
        j = i;
        while (j < n and equal(v[j], v[i])) {
            // process j
            if (above(v[j])) {
                if (v[j].color == colorAbove) score += v[j].value;
                else score -= v[j].value;
```

```
            }
            if (below(v[j]) ) {
                if (v[j].color == colorBelow) score += v[j].value;
                else score -= v[j].value;
            }
            j++;
        }
        if (score < ans) ans = score;
    }
    return ans;
}


int main(){
    int P, L;
    while (cin >> P >> L) {
        if (P == 0 and L == 0) break;
        for (int i = 0; i < P; ++i) {
            cin >> trees[i].x >> trees[i].y >> trees[i].value;
            trees[i].color = 0;
        }
        for (int i = 0; i < L; ++i) {
            cin >> trees[P + i].x >> trees[P + i].y >> trees[P + i].value;
            trees[P + i].color = 1;
        }
        int n = P + L;

        int ans = 1 << 30;
        for (int i = 0; i < n; ++i) {
            const Tree &pivot = trees[i];
            int m = 0;
            for (int j = 0; j < n; ++j) if (i != j) {
                v[m].x = trees[j].x - pivot.x;
                v[m].y = trees[j].y - pivot.y;
                v[m].color = trees[j].color;
```

```
                v[m].value = trees[j].value;
                m++;
            }
            int score;

            score = sweep(m, 0);
            if (score < ans) ans = score;
            score = sweep(m, 1);
            if (score < ans) ans = score;
        }
        cout << ans << endl;
    }

    return 0;
}
```