Universidad de Sevilla

Teorema del Sándwich de Ham

Kenny Flores, Pablo Dávila, Pablo Reina

2025-08-28

Índice

1. Preparación concurso

template.cpp

```cpp
//Librería estándar (Solo existe en Linux)
#include <bits/stdc++.h>
using namespace std;

#define rep(i, a, b) for(int i = a; i < (b); ++i)
#define all(x) begin(x), end(x)
#define sz(x) (int)(x).size()
typedef long long ll;
typedef pair<int, int> pii;
typedef vector<int> vi;

int main() {
  // Optimización E/S
  cin.tie(0)->sync_with_stdio(0);
  cin.exceptions(cin.failbit);
}
```

run.sh

```sh
# Usa este comando para compilar y ejecutar tu programa
# con entradas desde un archivo (ej: input.txt)
g++ -std=c++17 -O2 sol.cpp -o sol && ./sol < input.txt
```

troubleshoot.txt

```
Antes de enviar (Pre-submit)
- Escribe algunos casos de prueba simples si las muestras no son suficientes.
- ¿Los límites de tiempo están ajustados? Si es así, genera casos máximos.
- ¿El uso de memoria está bien?
- ¿Podría haber algún desbordamiento?
- Asegúrate de enviar el archivo correcto.

Respuesta incorrecta (Wrong answer)
- ¡Imprime tu solución! Imprime también la salida de depuración.
- ¿Estás limpiando todas las estructuras de datos entre casos de prueba?
- ¿Tu algoritmo puede manejar todo el rango de entrada?
- Lee de nuevo el enunciado completo del problema.
- ¿Estás manejando correctamente todos los casos límite?
- ¿Has entendido bien el problema?
- ¿Alguna variable sin inicializar?
- ¿Algún desbordamiento?
- ¿Confusión entre N y M, i y j, etc.?
- ¿Seguro de que tu algoritmo funciona?
- ¿Qué casos especiales no has considerado?
- ¿Seguro de que las funciones de la STL que usas funcionan como piensas?
- Añade algunas aserciones, quizá vuelve a enviar.
- Crea algunos casos de prueba para ejecutar tu algoritmo.
- Recorre el algoritmo con un caso sencillo.
- Vuelve a revisar esta lista.
- Explica tu algoritmo a un compañero.
- Pídele a tu compañero que revise tu código.
- Da un pequeño paseo, por ejemplo al baño.
- ¿El formato de salida es correcto? (incluyendo espacios en blanco)
- Reescribe tu solución desde cero o deja que un compañero lo haga.

Error en tiempo de ejecución (Runtime error)
- ¿Has probado todos los casos límite de forma local?
- ¿Alguna variable sin inicializar?
- ¿Estás leyendo o escribiendo fuera del rango de algún vector?
- ¿Alguna aserción que pueda fallar?
- ¿Alguna posible división por 0? (por ejemplo mod 0)
- ¿Alguna recursión infinita posible?
- ¿Punteros o iteradores invalidados?
- ¿Estás usando demasiada memoria?
- Depura con reenvíos (por ejemplo, señales remapeadas, ver *Various*).

Límite de tiempo excedido (Time limit exceeded)
- ¿Tienes algún bucle infinito posible?
- ¿Cuál es la complejidad de tu algoritmo?
- ¿Estás copiando muchos datos innecesarios? (usa referencias)
- ¿Qué tan grande es la entrada y la salida? (considera `scanf`)
- Evita `vector`, `map`. (usa arreglos / `unordered_map`)
- ¿Qué opinan tus compañeros sobre tu algoritmo?

Límite de memoria excedido (Memory limit exceeded)
- ¿Cuál es la cantidad máxima de memoria que debería necesitar tu algoritmo?
- ¿Estás limpiando todas las estructuras de datos entre casos de prueba?
```

2. Matemáticas

gcd.h

```cpp
// Máximo común divisor (GCD)
int gcd(int a, int b) {
    while (b) {
        int t = b;
        b = a % b;
        a = t;
    }
    return a;
}
```

lcm.h

```cpp
// Mínimo común múltiplo (LCM)
int lcm(int a, int b) {
    return a / gcd(a, b) * b;  // evitar overflow
}
```

sieve.h

```cpp
// Criba de Erastótenes: Algoritmo para generar
// números primos. Gracias a noahdris (UCppM)
int N = 30;
vector<bool> es_primo(N+1,true);
vector<int> primos;
for(int i = 2; i <= N; i++){
    if(es_primo[i]){
        primos.push_back(i);
```

```
        for(int j = i; j*i <= N; j++) es_primo[j*i] = false;
    }
}
```

binary-exp.h

```
//Exponenciación rápida
//Calcula $a^b mod m$ de manera eficiente.
// https://cp-algorithms.com/algebra/binary-exp.html
long long binpow(long long a, long long b, long long m) {
    a %= m;
    long long res = 1;
    while (b > 0) {
        if (b & 1)
            res = res * a % m;
        a = a * a % m;
        b >>= 1;
    }
    return res;
}
```

Coeficientes Multinomiales

EL siguiente código calcula: $\binom{k_1+\ldots+k_n}{k_1,k_2,\ldots,k_n} = \frac{\sum_{k_i}}{k_1!k_2!\ldots k_n!}$

multinomial.h

```
ll multinomial(vi& v) {
    ll c = 1, m = v.empty() ? 1 : v[0];
    rep(i,1,sz(v)) rep(j,0,v[i]) c = c * ++m / (j+1);
    return c;
}
```

3. Estructuras de Datos

Implementaciones de estructuras de datos no estándar de la STL.

trie.h

```
// Trie construido en base
// al código de Los BoquerO(n³) UNED
class TrieNode {
public:
    std::unordered_map<char, TrieNode*> children;
    bool isEndOfWord;
    int count;

    TrieNode() : isEndOfWord(false), count(0) {}

    ~TrieNode() {
        for(auto& pair : children) {
            delete pair.second;
        }
    }
};

class Trie {
private:
    TrieNode* root;

public:
    Trie() {
        root = new TrieNode();
    }

    ~Trie() {
        delete root;
    }

    void insert(std::string word) {
        TrieNode* node = root;
        for (char c : word) {
                if (node->children.find(c) == node->children.end()) {
                node->children[c] = new TrieNode();
            }
            node = node->children[c];
            node->count++;
        }
        node->isEndOfWord = true;
    }

    bool search(std::string word) {
        TrieNode* node = root;
        for (char c : word) {
                if (node->children.find(c) == node->children.end()) {
                return false;
            }
            node = node->children[c];
        }
        return node != nullptr && node->isEndOfWord;
    }

    bool startsWith(std::string prefix) {
        TrieNode* node = root;
        for (char c : prefix) {
                if (node->children.find(c) == node->children.end()) {
                return false;
            }
            node = node->children[c];
        }
        return true;
    }

    int countWordsStartingWith(std::string prefix) {
        TrieNode* node = root;
        for (char c : prefix) {
                if (node->children.find(c) == node->children.end()) {
                return 0; // Prefix not found
            }
            node = node->children[c];
        }
        return node->count;
    }
};
```

segment_tree.h

```
// Es una estructura empleada para optimizar
// operaciones sobre rangos (segmentos) de un array.
// Gracias a Dalopir (UCppM)

// Funciones para configurar el segment tree
template<typename T> struct Min {
    T neutral = INT_MAX;
    T operator()(T x, T y) { return min(x, y); }
    T rep(T x, int c) { return x; }
};

template<typename T> struct Max {
    T neutral = INT_MIN;
    T operator()(T x, T y) { return max(x, y); }
    T rep(T x, int c) { return x; }
};
```

```cpp
template<typename T> struct Sum {
    T neutral = 0;
    T operator()(T x, T y) { return x+y; }
    T inv(T x) { return -x; }
    T rep(T x, int c) { return x*c; }
};

template<typename T> struct Mul {
    T neutral = 1;
    T operator()(T x, T y) { return x*y; }
    T inv(T x) { return 1/x; }
    T rep(T x, int c) { return pow(x, c); }
};

// Configuracion del segment tree
F para las queries, G para las actualizaciones
```cpp
template<typename T> struct STOP {
    using F = Max<T>; using G = Sum<T>;
    // d(g(a, b, ...), x, c): Distribute g over f
    // Ex: max(a+x, b+x, ...) = max(a, b, ...)+x
    // Ex: sum(a+x, b+x, ...) = sum(a, b, ...)+x*c
    static T d(T v, T x, int c) { return G()(v, x); }
};

// Segment Tree Básico
template<typename T, typename OP = STOP<T>> struct
ST {
    typename OP::F f; typename OP::G g;
    ST *L = 0, *R = 0; int l, r, m; T v;

    ST(const vector<T> &a, int ql, int qr)
    : l(ql), r(qr), m((l+r)/2) {
        if (ql == qr) v = a[ql];
          else L = new ST(a, ql, m), R = new ST(a,
m+1, qr),
             v = f(L->v, R->v);
    }
    ST(const vector<T> &a) : ST(a, 0, a.size()-1) {}
    ~ST() { delete L; delete R; }

    T query(int ql, int qr) {
        if (ql <= l && r <= qr) return v;
        if (r < ql || qr < l) return f.neutral;
        return f(R->query(ql, qr), L->query(ql, qr));
    }
    void apply(int i, T x) {
        if (l == r) { v = g(x, v); return; }
        if (i <= m) L->apply(i, x);
        else        R->apply(i, x);
        v = f(L->v, R->v);
    }
    void set(int i, T x) {
        if (l == r) { v = x; return; }
        if (i <= m) L->set(i, x);
        else        R->set(i, x);
        v = f(L->v, R->v);
    }

    T get(int i) { return query(i, i); }
};
```

4. Grafos

Un grafo G=(V,E) es un conjunto de vértices V y aristas (E, que almacena la información de conectividad entre los vértices en V).

Dijkstra: Se utiliza para encontrar el camino más corto desde un nodo de inicio hasta todos los demás nodos en un grafo ponderado.

Dijkstra.h

```cpp
const int INF = 1000000000;
vector<vector<pair<int, int>>> adj;

void dijkstra(int s, vector<int> & d, vector<int> &
p) {
    int n = adj.size();
    d.assign(n, INF);
    p.assign(n, -1);
    vector<bool> u(n, false);

    d[s] = 0;
    for (int i = 0; i < n; i++) {
        int v = -1;
        for (int j = 0; j < n; j++) {
            if (!u[j] && (v == -1 || d[j] < d[v]))
                v = j;
        }

        if (d[v] == INF)
            break;

        u[v] = true;
        for (auto edge : adj[v]) {
            int to = edge.first;
            int len = edge.second;

            if (d[v] + len < d[to]) {
                d[to] = d[v] + len;
                p[to] = v;
            }
        }
    }
}
```

DFS: Recorre todos los nodos de un grafo o árbol profundizando en cada rama antes de retroceder.

dfs.h

```cpp
vector<vector<int>> adj; // graph represented as an
adjacency list
int n; // number of vertices

vector<bool> visited;

void dfs(int v) {
    visited[v] = true;
    for (int u : adj[v]) {
        if (!visited[u])
            dfs(u);
    }
}
```

BFS: Recorre todos los nodos de un grafo o árbol nivel por nivel.

bfs.h

```cpp
vector<vector<int>>  adj;     //  adjacency  list
representation
int n; // number of nodes
int s; // source vertex

queue<int> q;
vector<bool> used(n);
```
```

```
vector<int> d(n), p(n);

q.push(s);
used[s] = true;
p[s] = -1;
while (!q.empty()) {
    int v = q.front();
    q.pop();
    for (int u : adj[v]) {
        if (!used[u]) {
            used[u] = true;
            q.push(u);
            d[u] = d[v] + 1;
            p[u] = v;
        }
    }
}
```

Bellman-Ford: Calcula los caminos más cortos desde s en un grafo que puede tener aristas con pesos negativos.

- Los nodos inalcanzables obtienen `dist = inf`; los nodos alcanzables a través de ciclos de peso negativo obtienen `dist = -inf`.
- Se asume que $V^2 \cdot \max|w_i| < 2^{63}$.

BellmanFord.h

```
#pragma once

const ll inf = LLONG_MAX;
struct Ed { int a, b, w, s() { return a < b ? a : -a; }};
struct Node { ll dist = inf; int prev = -1; };

void bellmanFord(vector<Node>& nodes, vector<Ed>& eds, int s) {
  nodes[s].dist = 0;
   sort(all(eds), [](Ed a, Ed b) { return a.s() < b.s(); });

  int lim = sz(nodes) / 2 + 2; // /3+100 with shuffled vertices
  rep(i,0,lim) for (Ed ed : eds) {
    Node cur = nodes[ed.a], &dest = nodes[ed.b];
    if (abs(cur.dist) == inf) continue;
    ll d = cur.dist + ed.w;
    if (d < dest.dist) {
      dest.prev = ed.a;
      dest.dist = (i < lim-1 ? d : -inf);
    }
  }
  rep(i,0,lim) for (Ed e : eds) {
    if (nodes[e.a].dist == -inf)
      nodes[e.b].dist = -inf;
  }
}
```

FloydWarshall: Calcula las distancias más cortas entre todos los pares de un grafo dirigido que podría tener aristas con pesos negativos.

- La entrada es una matriz de distancias m, donde m[i][j] = inf si i y j no son adyacentes.
- Como salida, m[i][j] se establece en la distancia más corta entre i y j, inf si no existe camino, o -inf si el camino pasa por un ciclo de peso negativo.

FloydWarshall.h

```
#pragma once

const ll inf = 1LL << 62;
void floydWarshall(vector<vector<ll>>& m) {
  int n = sz(m);
  rep(i,0,n) m[i][i] = min(m[i][i], 0LL);
  rep(k,0,n) rep(i,0,n) rep(j,0,n)
    if (m[i][k] != inf && m[k][j] != inf) {
      auto newDist = max(m[i][k] + m[k][j], -inf);
      m[i][j] = min(m[i][j], newDist);
    }
  rep(k,0,n) if (m[k][k] < 0) rep(i,0,n) rep(j,0,n)
    if (m[i][k] != inf && m[k][j] != inf) m[i][j] = -inf;
}
```

EdmondsKarp: Algoritmo de flujo con complejidad garantizada $O(VE^2)$. Para obtener los valores de flujo de las aristas, compara las capacidades antes y después, y toma solo los valores positivos.

EdmondsKarp.h

```
#pragma once

template<class T> T
edmondsKarp(vector<unordered_map<int, T>>&
    graph, int source, int sink) {
  assert(source != sink);
  T flow = 0;
  vi par(sz(graph)), q = par;

  for (;;) {
    fill(all(par), -1);
    par[source] = 0;
    int ptr = 1;
    q[0] = source;

    rep(i,0,ptr) {
      int x = q[i];
      for (auto e : graph[x]) {
        if (par[e.first] == -1 && e.second > 0) {
          par[e.first] = x;
          q[ptr++] = e.first;
          if (e.first == sink) goto out;
        }
      }
    }
    return flow;
out:
    T inc = numeric_limits<T>::max();
    for (int y = sink; y != source; y = par[y])
      inc = min(inc, graph[par[y]][y]);

    flow += inc;
    for (int y = sink; y != source; y = par[y]) {
      int p = par[y];
     if ((graph[p][y] -= inc) <= 0) graph[p].erase(y);
      graph[y][p] += inc;
    }
  }
}
```

5. Geometría

point.h

```
// Clase para manejar puntos en el plano.
// T puede ser, por ejemplo, double o long long.
// Evite int. Sacado de KACTL
```

```cpp
#pragma once

template <class T> int sgn(T x) { return (x > 0) - (x < 0); }
template<class T>
struct Point {
  typedef Point P;
  T x, y;
  explicit Point(T x=0, T y=0) : x(x), y(y) {}
   bool operator<(P p) const { return tie(x,y) < tie(p.x,p.y); }
      bool operator==(P p) const { return tie(x,y)==tie(p.x,p.y); }
  P operator+(P p) const { return P(x+p.x, y+p.y); }
  P operator-(P p) const { return P(x-p.x, y-p.y); }
  P operator*(T d) const { return P(x*d, y*d); }
  P operator/(T d) const { return P(x/d, y/d); }
  T dot(P p) const { return x*p.x + y*p.y; }
  T cross(P p) const { return x*p.y - y*p.x; }
  T cross(P a, P b) const { return (a-*this).cross(b-*this); }
  T dist2() const { return x*x + y*y; }
 double dist() const { return sqrt((double)dist2()); }
  // angle to x-axis in interval [-pi, pi]
  double angle() const { return atan2(y, x); }
   P unit() const { return *this/dist(); } // makes dist()=1
   P perp() const { return P(-y, x); } // rotates +90 degrees
  P normal() const { return perp().unit(); }
   // returns point rotated 'a' radians ccw around the origin
  P rotate(double a) const {
    return P(x*cos(a)-y*sin(a),x*sin(a)+y*cos(a)); }
  friend ostream& operator<<(ostream& os, P p) {
    return os << "(" << p.x << "," << p.y << ")"; }
};
```

Envolvente Convexa (Convex Hull)

Devuelve un vector con los puntos del envolvente convexo en orden antihorario. Los puntos que se encuentran en el borde de la envolvente entre otros dos puntos no se consideran parte de la envolvente

ConvexHull.h

```cpp
#pragma once
#include "Point.h"
typedef Point<ll> P;
vector<P> convexHull(vector<P> pts) {
  if (sz(pts) <= 1) return pts;
  sort(all(pts));
  vector<P> h(sz(pts)+1);
  int s = 0, t = 0;
  for (int it = 2; it--; s = --t, reverse(all(pts)))
    for (P p : pts) {
      while (t >= s + 2 && h[t-2].cross(h[t-1], p) <= 0) t--;
      h[t++] = p;
    }
  return {h.begin(), h.begin() + t - (t == 2 && h[0] == h[1])};
}
```

6. Apéndices

techniques.txt

```
Recursión
  Dividir y conquistar
    Encontrar puntos interesantes en $N \log N$
Análisis de algoritmos
  Teorema maestro
  Complejidad temporal amortizada
Algoritmo voraz (Greedy)
  Planificación
  Suma máxima de subvector contiguo
  Invariantes
  Codificación Huffman
Teoría de grafos
  Grafos dinámicos (mantenimiento extra)
  Búsqueda en anchura (BFS)
  Búsqueda en profundidad (DFS)
    * Árboles normales / Árboles DFS
  Algoritmo de Dijkstra
  AGM: Algoritmo de Prim
  Bellman-Ford
  Teorema de König y cobertura de vértices
  Flujo máximo de costo mínimo
  Conmutación de Lovász
  Teorema de matrices de árboles
  Emparejamiento máximo, grafos generales
  Hopcroft-Karp
  Teorema de matrimonio de Hall
  Secuencias gráficas
  Floyd-Warshall
  Ciclos de Euler
  Redes de flujo
    * Caminos aumentantes
    * Edmonds-Karp
  Emparejamiento bipartito
  Cobertura mínima de caminos
  Ordenamiento topológico
  Componentes fuertemente conectados
  2-SAT
   Vértices de corte, aristas de corte y componentes biconectadas
  Coloración de aristas
    * Árboles
  Coloración de vértices
    * Grafos bipartitos (=> árboles)
    * $3^n$ (caso especial de set cover)
  Diámetro y centroide
  $k$-ésimo camino más corto
  Ciclo más corto
Programación dinámica
  Mochila
  Cambio de monedas
  Subsecuencia común más larga
  Subsecuencia creciente más larga
  Número de caminos en un DAG
  Camino más corto en un DAG
  Programación dinámica sobre intervalos
  Programación dinámica sobre subconjuntos
  Programación dinámica sobre probabilidades
  Programación dinámica sobre árboles
  $3^n$ set cover
  Dividir y conquistar
  Optimización de Knuth
  Optimización de convex hull
  RMQ (Sparse Table o saltos $2^k$)
  Ciclo bitónico
   Particionamiento logarítmico (loop sobre el más restringido)
Combinatoria
```

- Cálculo de coeficientes binomiales
- Principio del palomar
- Inclusión/exclusión
- Números de Catalan
- Teorema de Pick

Teoría de números

- Partes enteras
- Divisibilidad
- Algoritmo euclidiano
- Aritmética modular
  - Multiplicación modular
  - Inversos modulares
  - Exponenciación modular por cuadrados
- Teorema chino del resto
- Pequeño teorema de Fermat
- Teorema de Euler
- Función Phi
- Número de Frobenius
- Reciprocidad cuadrática
- Pollard-Rho
- Miller-Rabin
- Levantamiento de Hensel
- Saltos de raíces de Vieta

Teoría de juegos

- Juegos combinatorios
- Árboles de juego
- Minimax
- Nim
- Juegos sobre grafos
- Juegos sobre grafos con ciclos
- Números de Grundy
- Juegos bipartitos sin repetición
- Juegos generales sin repetición
- Poda alfa-beta

Teoría de la probabilidad

Optimización

- Búsqueda binaria
- Búsqueda ternaria
- Unimodalidad y funciones convexas
- Búsqueda binaria sobre derivadas

Métodos numéricos

- Integración numérica
- Método de Newton
- Búsqueda de raíces con búsqueda binaria/ternaria
- Búsqueda de la sección áurea

Matrices

- Eliminación gaussiana
- Exponenciación por cuadrados

Ordenamiento

- Radix sort

Geometría

- Coordenadas y vectores
  - Producto cruz
  - Producto escalar
- Convex hull
- Corte de polígonos
- Par más cercano
- Compresión de coordenadas
- Quadtrees
- KD-trees
- Todas las intersecciones segmento-segmento

Barrido (Sweeping)

- Discretización (convertir en eventos y barrer)
- Barrido por ángulos
- Barrido de líneas
- Segundas derivadas discretas

Cadenas (Strings)

- Subcadena común más larga
- Subsecuencias palíndromas
- Knuth-Morris-Pratt
- Tries
- Hashes polinomiales rodantes
- Array de sufijos
- Árbol de sufijos
- Aho-Corasick
- Algoritmo de Manacher
- Listas de posiciones de letras

Búsqueda combinatoria

- Meet in the middle
- Fuerza bruta con poda
- Mejor primero (A*)
- Búsqueda bidireccional
- DFS/A* con profundización iterativa

Estructuras de datos

- LCA (saltos 2^k en árboles en general)
- Técnica pull/push en árboles
- Descomposición heavy-light
- Descomposición por centroides
- Propagación perezosa (Lazy propagation)
- Árboles auto-balanceados
- Convex hull trick (wcipeg.com/wiki/Convex_hull_trick)
- Colas/Stacks monotónicos / colas deslizantes
- Cola deslizante usando 2 stacks
- Segment tree persistente