Universidad de Sevilla

Teorema del Sándwich de Ham

Kenny Flores, Pablo Dávila, Pablo Reina

2025-05-16

Índice

1. Setup

Cuando trabajas en la terminal, usa este comando para compilar y ejecutar tu programa con entradas desde un archivo (ej: input.txt)

```
g++ main.cpp -o main && ./main < input.txt
```

1.1. Librería estándar

Es un header no estándar que incluye todas las bibliotecas estándar de C++ de una sola vez.

```
#include <bits/stdc++.h>
```

1.2. Lectura línea a línea

```
std::string line;
while(std::getline(std::cin, line)){ ... }
```

1.3. Lectura número a número

```
int n;
while(std::cin >> n){ ... }
```

1.4. Lectura carácter a carácter

No ignora espacios ni saltos de línea

```
char c;
while(std::cin::get(c)){ ... }
```

1.5. Optimización de E/S

En problemas con entradas/salidas grandes (ej.: programación competitiva), estas líneas aceleran `cin/cout`:

```
int main() {
    // Desincroniza C++ de C (¡cin/cout más rápidos!)
    std::ios_base::sync_with_stdio(false);
    // Evita que cin vacíe cout antes de cada lectura
    std::cin.tie(nullptr);
    // tu código...
}
```

2. Operaciones con cadenas

2.1. split

```
std::vector<std::string> split(const std::string &
str, char delimiter){
    std::vector<std::string> tokens;
    std::stringstream ss(str);
    std::string token;
    while(std::getline(ss, token, delimiter)){
        if(!token.empty()) {tokens.push_back(token); }
    }
    return tokens;
}
```

2.2. concatenación

```
std::stringstream msg;
msg << "hola";
std::cout << msg.str() << "\n";
```

2.3. Búsqueda subcadena

```
std::string str = "Hello World!";
if(str.find("World") != std::string::npos) { /*
Exists*/}
```

2.4. Parseo entero

```
int num =  std::stoi(str);
long num = std::stol(str);
long long num = std::stoll(str);
```

2.5. Parseo flotante

```
float num = std::stof(str);
double num = std::stod(str);
long double num = std::stold(str)
```

2.6. Parseo múltiples números

```
std::istringstream iss("123 456");
int num1, num2;
iss >> num1 >> num2; // num1 = 123, num2=456
```

3. Estructuras de Datos

3.1. Segment Tree

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

3.1.1. 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; }
};

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); }
};
```

3.1.2. Configuracion del segment tree

F para las queries, G para las actualizaciones

```
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); }
};
```

3.1.3. 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. Algoritmos de Búsqueda y ordenamiento

4.1. Búsqueda binaria

Busca en una lista ordenada dividiéndola a la mitad en cada paso. La complejidad en tiempo es de O(log n).

4.1.1. Comprobar si existe elemento

```
bool exists = binary_search(vec.begin(), vec.end(),
value);
```

4.1.2. Posición del elemento

```
auto it = lower_bound(vec.begin(), vec.end(), value);
if (it != vec.end() && *it == value) {
  int index = distance(vec.begin(), it); // posición
del elemento
  cout << "Posición exacta: " << index << '\n';

}
```

4.1.3. Elemento más pequeño que verifica ser más grande que value

```
it = upper_bound(vec.begin(), vec.end(), value);
if (it != vec.end()) {
  int index = distance(vec.begin(), it);
  cout << "Más pequeño que es mayor a value: " <<
index << '\n';
}
```

4.1.4. Elemento más pequeño que verifica ser mayor o igual a «value»

```
it = lower_bound(vec.begin(), vec.end(), value);
  if (it != vec.end()) {
    int index = distance(vec.begin(), it);
     cout << "Más pequeño que es mayor o igual a
value: " << index << '\n';
    }
```

4.1.5. Elemento más grande menor que «value»

```
it = lower_bound(vec.begin(), vec.end(), value);
  if (it != vec.begin()) {
    --it; // retrocede al elemento anterior
    int index = distance(vec.begin(), it);
    cout << "Más grande menor que value: " << index <<
'\n';
}
```

4.1.6. Elemento más grande menor o igual que «value»

```
it = upper_bound(vec.begin(), vec.end(), value);
if (it != vec.begin()) {
  --it; // retrocede al elemento anterior
  if (*it <= value) {
    int index = distance(vec.begin(), it);
    cout << "Más grande menor o igual que value: " <<
index << '\n';
  }
}
```

5. Grafos

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

5.1. Algoritmo de Dijkstra

Se utiliza para encontrar el camino más corto desde un nodo de inicio hasta todos los demás nodos en un grafo ponderado, con el objetivo de resolver problemas como rutas más eficientes en redes, mapas de carreteras o en la optimización de costos de conexión.

```
int main() {
    int V, E;
    cin >> V >> E;

    vector<vector<pair<int, int>>> edges(V);
    for (int i = 0; i < E; i++) {
        int u, v, w;
        cin >> u >> v >> w;
        edges[u].push_back({v, w});
        edges[v].push_back({u, w});
    }

    vector<int> dist(V, INT_MAX);
    vector<bool> used(V, false);
    priority_queue<pair<int, int>> pq;

    dist[0] = 0, pq.push({0, 0});
    while (!pq.empty()) {
        auto [uw, u] = pq.top(); pq.pop(); uw *= -1;
        if (used[u]) continue; used[u] = true;

        for (auto [v, w] : edges[u]) {
            int vw = uw+w;
```

```
            if (vw < dist[v])
                dist[v] = vw, pq.push({-vw, v});
        }
    }

    cout << dist[V-1] << "\n";
}
```

5.2. Algoritmo de Floyd-Warshall

Es útil cuando necesitas calcular las distancias más cortas entre todos los pares de vértices en un grafo. En general no es buena idea utilizarlo cuando $450 < |V|$.

```
//TODO
```

5.3. Búsqueda en Profundidad (DFS)

5.4. Búsqueda en Amplitud (BFS)

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

```
// TODO
```

```
//TODO:
```

6. Matemáticas

6.1. Máximo común divisor (GCD)

```
int gcd(int a, int b) {
    while (b) {
        int t = b;
        b = a % b;
        a = t;
    }
    return a;
}
```

6.2. Mínimo común múltiplo (LCM)

```
int lcm(int a, int b) {
    return a / gcd(a, b) * b;  // evitar overflow
}
```

6.3. Criba de Erastótenes

Algoritmo para generar números primos

```
// Proporcionado por 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;
    }
}
```

6.4. Exponenciación rápida

Calcula $a^b \bmod m$ de manera eficiente.

```
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;
}
```

6.5. Coeficientes binomiales

El coeficiente binomial $\binom{n}{k}$ representa el número de formas de elegir un subconjunto de k elementos de un conjunto de n elementos. Por ejemplo $\binom{4}{2} = 6$ porque el conjunto $\{1, 2, 3, 4\}$ tiene 6 subconjuntos de 2 elementos $\{1, 2\}, \{1, 3\}, \{1, 4\}, \{2, 3\}, \{2, 4\}, \{3, 4\}$. Para calcular estos coeficientes se utiliza la fórmula $\binom{n}{k} = \frac{n!}{k!(n-k)!}$.

```
// TODO
```

7. Strings

7.1. Trie

```
// 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;
    }
};
```

8. Geometría

8.1. Clase Punto

```
// Proporcionado por (UCppM)
struct pt{
  uds x,y;
  pt(){}
  pt(uds x, uds y): x(x), y(y){}

  pt operator+(const pt &p) const {return {x + p.x, y + p.y};}
  pt operator*(uds k) const {return {k*x, k*y};}
  pt operator-() const { return *this *-1; }
  pt operator-(const pt &p) const {return *this + (-p);}
  pt operator/(uds d) const {
    if (eq(d, 0)) throw runtime_error("operator/ Recuerdos de vietADAMan (p/0)");
    return *this*(1/d);
  }

  pt& operator+=(const pt& p) { return *this = *this+p; };
  pt& operator-=(const pt& p) { return *this = *this-p; };
  pt& operator*=(uds k) { return *this = *this*k; };
  pt& operator/=(uds k) { return *this = *this/k; };

  bool operator==(const pt &a) const { return eq(a.x, x) && eq(a.y,y); }
  bool operator!=(const pt &a) const { return !(a == *this); }
  bool operator>(const pt &a) const { return (eq(a.x, x) ? lt(a.y, y): lt(a.x, x));}
  bool operator<(const pt &a) const { return a > *this; }
  pt rot() const {return pt(-y, x);} // rotate 90 degrees counter-clockwise
};
ostream& operator<<(ostream &out, const pt &p)
{ return out<<p.x<<" "<<p.y;}
pt operator*(uds k, const pt &p) { return p*k; }
uds dot(const pt &p, const pt &q) { return p.x*q.x + p.y*q.y; }
uds norma(const pt &p) { return sqrt(dot(p,p)); }
uds angle(const pt &p, const pt &q) {return acos(dot(p,q)/(norma(p)*norma(q)));}
uds cross(const pt &p, const pt &q) {return p.x*q.y - p.y*q.x;}

bool collinear(pt a, pt b) { return eq(cross(a, b), 0); } //si son li

uds orient(pt a, pt b, pt c) {return cross(b-a, c-a);} //
int orientS(pt a, pt b, pt c) {return sgn(orient(a, b, c)); }

bool cw(pt a, pt b, pt c, bool in_colli) {
  int o = orientS(a, b, c); return o<0 || (in_colli && o==0); // o < 0
}
bool ccw(pt a, pt b, pt c, bool in_colli) {
  int o = orientS(a, b, c); return o>0 || (in_colli && o==0); // o > 0
}
bool collinear(pt a, pt b, pt c) { return eq(orient(a, b, c),0); } // ver si tres puntos estan alineados
```

8.2. Clase Linea

```
struct line{
  pt p, v;
  line(pt v): p(pt(0,0)), v(v){}
  line(pt p, pt v): p(p), v(v){}
  line(uds a, uds b, uds c): p((eq(b,0) ? pt(-c/a, 0) : pt(0, -c/b) )), v(pt(b,-a)){} // ax+by+c=0

  bool in_line(const pt &q) const {return collinear(p-q , v);}
  bool operator ==(const line &r) const {return in_line(r.p) && collinear(v, r.v);}
  bool operator !=(const line &r) const {return !(*this == r);}

  line perp(const pt &b) const {return {b, v.rot()};}
};

vector<pt> intersection(line l1, line l2) {
  if(l1 == l2) throw runtime_error("intersection() son la misma");
  uds d = cross(l1.v, l2.v), c1 = cross(l1.v,l1.p), c2 = cross(l2.v,l2.p);
  if (eq(d,0)) return {};
  return {(l2.v*c1-l1.v*c2)/d};
}

pt proj(const pt &p, const line &r){return intersection(r, r.perp(p))[0];}
pt proj(const line &r, const pt &p){return proj(p, r);}

uds dist(const pt &p, const pt &q){return norma(p-q);}
uds dist(const pt &p, const line &r){return dist(p, proj(p,r));}
uds dist(const line &r, const pt &p){return dist(p,r);}
uds dist(const line &r, const line &s){return (collinear(r.v,s.v) ? dist(r,s.p): 0);}
```

8.3. Clase Polígono

```
struct poly {
  vector<pt> pts;
  poly(){}
  poly(vector<pt> pts): pts(pts){}

  uds perim() const {
    uds result = 0;
    for(int i = 0; i < pts.size(); i++) result +=
norma(pts[i] - pts[ (i+1)% pts.size()]);
    return result;
  }

  bool isConvex() const {
    bool hasPos=false, hasNeg=false;
    for (int i=0, n=pts.size(); i<n; i++) {
      int o = orientS(pts[i], pts[(i+1)%n],
pts[(i+2)%n]);
      if (o > 0) hasPos = true;
      if (o < 0) hasNeg = true;
    }
    return !(hasPos && hasNeg);
  }

  uds area() const{
    uds result = 0;
    for(int i = 0; i<pts.size(); i++) result +=
cross(pts[i] , pts[ (i+1)% pts.size()]);
    return abs(result)/2;
  }
    //   Check if point belongs to the polygon in
n  //
  bool inPol(const pt &point) const { // linear
    if(pts.size() == 0) return false;
    uds sum = 0;
    for(int i = 0; i<pts.size(); i++){
      if(point == pts[i] || point ==
pts[(i+1)%pts.size()]) return true;
      sum += angle(pts[i]-point,
pts[(i+1)%pts.size()]-point)*(2*ccw(point, pts[i],
pts[(i+1)%pts.size()],false) -1);
    }
    return eq(abs(sum), 2*pi);
  }

  bool inConvex(pt &point, bool strict = true) const
{ // log
    int a = 1, b = pts.size() - 1, c;
    if (orientS(pts[0], pts[a], pts[b]) > 0) swap(a,
b);
    if (orientS(pts[0], pts[a], point) > 0 ||
orientS(pts[0], pts[b], point) < 0 ||
       (strict && (orientS(pts[0], pts[a], point) == 0
|| orientS(pts[0], pts[b], point) == 0)))
      return false;
    while (abs(a - b) > 1) {
      c = (a + b) / 2;
      if (orientS(pts[0], pts[c], point) > 0) b = c;
      else a = c;
    }
    return orientS(pts[a], pts[b], point) < 0 ?
true : !strict;
  }

  void convex_hull(bool in_colli = false) {
    pt p0 = *min_element(pts.begin(), pts.end());
    sort(pts.begin(), pts.end(), [&p0](const pt& a,
const pt& b) {
      int o = orientS(p0, a, b); return o<0 || (o==0
&& lt(dist(p0,a), dist(p0,b)));
    });
    if (in_colli) {
      int i = pts.size()-1;
      for(;i && collinear(p0, pts[i], pts.back());
i--);
      reverse(pts.begin()+i+1, pts.end());
    }
    vector<pt> st;
    for (int i = 0; i < pts.size(); i++) {
      while (st.size() > 1 && !cw(st[st.size()-2],
st.back(), pts[i], in_colli)) st.pop_back();
      st.push_back(pts[i]);
    }
    if (!in_colli && st.size() == 2 && st[0] == st[1])
st.pop_back();
    pts = st;
  }

};
uds pick(int I,int B){return I-1 + B*0.5;} // Teorema
de Pick. I interior y B borde
```

9. Técnicas

9.1. Recursión

9.1.1. Divide y vencerás

- Encontrar puntos interesantes en N log N

9.1.2. Algoritmo codicioso (Greedy)

- Planificación
- Máxima suma de subvector contiguo
- Invariantes
- Codificación Huffman

9.2. Teoría de grafos

- Grafos dinámicos (registro adicional)
- Búsqueda en amplitud
- Búsqueda en profundidad
  - Árboles normales / Árboles DFS
- Algoritmo de Dijkstra
- MST: Algoritmo de Prim
- Bellman-Ford
- Teorema de Konig y cobertura de vértices
- Flujo máximo de costo mínimo
- Conmutador de Lovasz
- Teorema del árbol de matriz
- Emparejamiento máximo, grafos generales
- Hopcroft-Karp
- Teorema del matrimonio de Hall
- Secuencias gráficas
- Floyd-Warshall
- Ciclos de Euler
- Flujo máximo
  - Caminos aumentantes
  - Edmonds-Karp
- Emparejamiento bipartito
- Cobertura mínima de caminos
- Ordenación topológica
- Componentes fuertemente conectados
- 2-SAT

- Vértices de corte, aristas de corte y componentes biconectados
- Coloreado de aristas
  - Árboles
- Coloreado de vértices
  - Grafos bipartitos (=> árboles)
  - 3^n (caso especial de cobertura de conjuntos)
- Diámetro y centroide
- K-ésimo camino más corto
- Ciclo más corto

9.3. 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
- Cobertura de conjunto 3^n
- Divide y vencerás
- Optimización de Knuth
- Optimización del cascarón convexo
- RMQ (tabla dispersa, también conocido como saltos 2^k)
- Ciclo bitónico
- Partición logarítmica (bucle sobre lo más restringido)

9.4. Combinatoria

- Cálculo de coeficientes binomiales

9.5. Teoría de números

- Partes enteras
- Divisibilidad
- Algoritmo de Euclides
- Aritmética modular
  - Multiplicación modular
  - Inversos modulares
  - Exponentiación modular por cuadrados
- Teorema del resto chino
- Teorema de Euler
- Función Phi

9.6. Optimización

- Búsqueda binaria
- Búsqueda ternaria

9.7. Geometría

- Coordenadas y vectores Producto cruzado Producto escalar
- Envolvente convexa
- Corte de polígonos
- Par más cercano
- Compresión de coordenadas
- Intersección de todos los segmentos

9.8. Strings

- Subcadena común más larga
- Subsecuencias palindrómicas
- Tries
- Algoritmo de Manacher
- Listas de posiciones de letras

9.9. Búsqueda combinatoria

- Encuentro en el medio
- Fuerza bruta con poda
- Mejor primero (A estrella)
- Búsqueda bidireccional
- Profundización iterativa DFS / A estrella