INF-221
Algoritmos y Complejidad
Documentación

Índice general

1 Consejos y consideraciones

1.1. Forma eficiente de leer un problema

- Leer la entrada y salida del problema.
- Leer el enunciado (eliminar información innecesaria en función del paso anterior).
- Obtener **observaciones** y **propiedades** del problema.
- Pensar en un algoritmo (si es menos de 10^8 operaciones, es válido).

1.2. Estimación de complejidades aceptadas en función de tamaño de entrada

n	Peor complejidad aceptada
$n <= 10$	$O(n!), O(n^7), O(n^6)$
$n <= 20$	$O(2^n \cdot n), O(n^5)$
$n <= 80$	$O(n^4)$
$n <= 400$	$O(n^3)$
$n <= 7500$	$O(n^2)$
$n <= 7 \cdot 10^4$	$O(n\sqrt{n})$
$n <= 5 \cdot 10^5$	$O(n \log n)$
$n <= 5 \cdot 10^6$	$O(n)$
$n > 10^8$	$O(\log n), O(1)$

1.3. Técnicas para encontrar propiedades de un problema

1. Resolver desde lo específico a lo general.
2. Confiar en tus capacidades y supuestos.
3. Si no se te ocurre nada en 60 minutos, sigue con otra cosa.

1.4. Consejos para C++

La librería más utilizada y que engloba todo lo necesario para problemas de programación competitiva es:

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

También se recomienda definir aliases para tipos de datos recurrentes con el propósito de agilizar la escritura. Por ejemplo, algunas definiciones pueden ser

```
#define ll long long
#define ld long double
#define vvi std::vector<std::vector<int>>
/*...*/
```

A modo de optimizar, hay configuraciones que se pueden utilizar al inicio de cada programa:

```
std::ios_base::sync_with_stdio(false);
std::cin.tie(NULL);
std::cout.tie(NULL);
```

Para pruebas rápidas de samples, es posible usar el mismo ejecutable junto a un archivo de texto con las entradas esperadas.

```
bashiee@benjitooo $ g++ program.cpp
bashiee@benjitooo $ ./a.out < sample.txt
```

1.5. Consejos para programación dinámica

Cuándo NO usar DP

- **NO hay subestructura óptima**: No hay subproblemas con soluciones óptimas.
- **No hay superposición de problemas**: No hay subproblemas que se resuelvan más de una vez.
- **Hay un greedy que funciona.**
- **Espacio/Tiempo excesivo para las dimensiones del problema.**
- **Problemas NP-Completos sin restricciones.**

Metodología sistemática

1. **¿Qué quiero calcular? (costos, maximizar, minimizar...).**
2. **¿De qué variables depende?**
3. **Calcular complejidad (Espacio y tiempo).**
4. **¿Es muy costoso? (optimizar, volver al paso 2).**

Patrones comúnes

- **DP en 1D:** Subproblemas dependen de estados anteriores inmediatos (fibonacci, coin, stairs...).
- **DP en 2D:** Subproblemas dependen de dos parámetros (knapsack, LCS...)

Errores comúnes

- **Olvidar restricciones (estados inválidos).**
- **Índices incorrectos (límites del problema).**
- **Estados insuficientes (No se considera toda la información).**

Consejos clave

- **Buscar opciones recursivas naturales:** ¿Qué pasaría si existiera una función mágica? ¿Cuál sería la última decisión por tomar?
- **Identificar patrones de repetición:** ¿Los mismos subproblemas aparecen más de una vez?
- **Definir un estado por completo:** ¿Qué información se necesita para definir un subproblema por completo?
- **Considerar casos bases:** ¿Cuándo termina la recursión?
- **Empezar con casos pequeños:** Resolver manualmente casos pequeños. Buscar patrones.
- **Pensar en los últimos elementos**: ¿Cómo se llegó a este estado?

2 Complejidad

2.1. Desempeño de algoritmos

Los algoritmos no tienen un rendimiento fijo para cualquier tipo de entrada. Generalmente, hay situaciones donde se comportan de mejor o peor manera dependiendo de la noción de la cual trabajan. En el análisis y diseño de algoritmos se tienen en cuenta los siguientes casos para evaluar el rendimiento de un algoritmo.

Mejor caso

Caso donde el algoritmo funciona de la mejor manera. Por ejemplo, el mejor caso de algunos algoritmos de ordenamiento se considera el que la entrada ya esté ordenada.

Caso promedio

Caso donde el algoritmo posee un tiempo de ejecución esperado dentro una distribución de entradas. Para comprobarlo, se necesita una hipótesis probabilística.

Peor caso

Caso donde el algoritmo trabaja de peor forma. Es la situación que se utiliza generalmente para las notaciones asíntoticas. Un ejemplo es una entrada inversamente ordenada para los algoritmos de ordenamiento.

Caso amortizado

Costo promedio de **una operación** en una secuencia larga de operaciones que no depende de la distribución de la entrada. Un ejemplo es el método *push_back* para la clase *vector* de *C++*.

2.2. Invariante de ciclo

Una técnica que se utiliza a menudo para demostrar la correctitud de un algoritmo es usar un **invariante de ciclo.** Un *invariante* es una afirmación o predicado sobre un conjunto de variables relacionadas que es **siempre verdadero**. La idea es que este invariante sea **constante** en cada iteración y que explique la relación entre las variables de entrada y salida del ciclo.

Etapas del invariante de ciclo

Inicialización: El invariante debe ser verdadero **antes de empezar** el ciclo (caso base).

Mantención: El invariante es verdadero **antes de cada iteración del ciclo** y se mantiene verdadero hasta **antes de la siguiente iteración.** A menudo se utiliza **inducción** para probar este punto con una *hipótesis inductiva* para luego realizar un *paso inductivo*.

Finalización: El invariante debe ser verdadero **después de terminar** el ciclo.

Conclusión: Si el invariante es verdadero en cada etapa, el algoritmo es correcto.

2.3. Análisis asintótico

Una manera de medir la complejidad de un algoritmo es a través del *análisis asintótico*. Dentro de las dimensiones estudiadas, se consideran dos grandes aspectos:

- Complejidad temporal ($T(n)$): Predicción de operaciones fundamentales en base al tamaño de la entrada.
- Complejidad espacial ($E(n)$): Predicción del costo en memoria en base al tamaño de la entrada.

En base a estos criterios, se definen las siguientes notaciones:

Notación Big O

La notación *Big O* se relaciona con $T(n)$.

Se utiliza $O(f(n))$ como representación del límite en el peor caso de tiempo de ejecución de un algoritmo en base a un tamaño de entrada n.

Formalmente, se define como:

$$T(n) = O(f(n)) \quad ssi \quad \exists c, n_0 > 0 \quad t.q \quad T(n) \leq c \cdot f(n), \forall n \geq n_0$$

Notación little o

La notación *little o* se relaciona con $T(n)$.

A diferencia de *Big O*, la notación *little o* permite afirmar que $T(n)$ **siempre** es menor que $c \cdot f(n)$ por muy pequeño que sea c a partir de un n_0 y puede hacerse más pequeña de manera arbitraria.

Formalmente, se define como:

$$T(n) = o(f(n)) \quad ssi \quad \exists c, n_0 > 0 \quad t.q \quad T(n) < c \cdot f(n), \forall n \geq n_0$$

Por otro lado, también se puede intuir a través de esta relación:

$$\lim_{n \to \infty} \left(\frac{T(n)}{f(n)} \right) = 0$$

Notación Big Omega

La notación *Big Omega* se relaciona con $T(n)$.

Se utiliza $\Omega(f(n))$ como representación de la cota inferior del tiempo de ejecución de un algoritmo para un tamaño de entrada n.

Formalmente, se define como:

$$T(n) = \Omega(f(n)) \quad ssi \quad \exists c, n_0 > 0 \quad t.q \quad T(n) \geq c \cdot f(n), \forall n \geq n_0$$

Notación little omega

La notación *litte omega* se relaciona con $T(n)$.

Similar a la notación *little o*, la notación *little omega* permite afirmar que $T(n)$ es **estrictamente** mayor que $c \cdot f(n)$ a partir de un n_0 sea cual sea el valor de c.

Formalmente, se define como:

$$T(n) = \omega(f(n)) \quad ssi \quad \exists c, n_0 > 0 \quad t.q \quad T(n) > c \cdot f(n), \forall n \geq n_0$$

Paralelamente, se puede intuir a partir de la siguiente relación:

$$\lim_{n \to \infty} \left(\frac{T(n)}{f(n)} \right) = \infty$$

Notación Big Theta

La notación *Big Theta* se relaciona con $T(n)$.

Se utiliza $\Theta(f(n))$ para un $T(n)$ donde se cumple que $T(n) = \emptyset(f(n))$ y $T(n) = \Omega(f(n))$ para simbolizar la cota superior e inferior en tiempo de ejecución de un algoritmo frente a un tamaño de entrada n.

Formalmente, se define como:

$$T(n) = \Theta(f(n)) \quad ssi \quad \exists c_1, c_2, n_0 > 0 \quad t.q \quad c_1 \cdot f(n) \leq T(n) \leq c_2 \cdot f(n), \forall n \geq n_0$$

2.4. Teorema maestro

Para determinar la complejidad de un algoritmo recursivo se utiliza a menudo el **teorema maestro**.

Formalmente, se define la siguiente relación para casos grandes:

$$T(n) \leq a \cdot T\left(\frac{n}{b}\right) + O(n^d)$$

Parámetros1:
a es el número de llamadas recursivas.
b el factor de disminución del tamaño de entrada (sub-problemas).
d el exponente asociado al trabajo no recursivo.

De esta relación y si se cumple que $a \geq 1, b > 1 y d \geq 0$, se cumple que:

$$T(n) = \begin{cases} O(n^d \log n) & \text{si } a = b^d \\ O(n^d) & \text{si } a < b^d \\ O(n^{\log_b a}) & \text{si } a > b^d \end{cases}$$

3 Estructuras de datos

3.1. Vector

El vector es un tipo de arreglo dinamico en C++. Aqui se encuentran sus operaciones principales

```
/**
* Creación de un vector
*/
std::vector<T> vector;

/**
* Creación de un vector con tamaño n
* @timecomplexity: O(n)
*/
std::vector<T> vector(n);

/**
* Creación de un vector con tamaño n y sus valores en 0
* @timecomplexity: O(n)
*/
std::vector<T> vector(n,0);

/**
* Retorna un iterador al inicio del vector
*/
vector.begin();

/**
* Retorna un iterador al final del vector
*/
vector.end();

/**
* Acceso a un elemento especifico de un vector
* @timecomplexity: O(1)
*/
vector[i];

/**
* Agregar un elemento al final del vector
* @timecomplexity: O(1)*
*/
vector.push_back(i);

/**
* Quitar un elemento del final del vector
* @timecomplexity: O(1)
*/
```

```
vector.pop_back();

/**
* Inserta un elemento en la posición i
* @timecomplexity: O(n)
*/
vector.insert(vector.begin()+i, item);

/**
* Elimina un elemento en la posición i
* @timecomplexity: O(n)
*/
vector.erase(vector.begin()+i, item);

/**
* Revisa si el vector esta vacio
* @timecomplexity: O(1)
*/
vector.empty();

/**
* Retorna el tamaño del vector
* @timecomplexity: O(1)
*/
vector.size();
```

3.2. Stack

La Stack es la estructura de datos que cumple con la propiedad de LIFO

```
/**
* Declaración de un stack con tipo de dato T
*/
std::stack<T> st;

/**
* Retorna el tamaño del stack
*/
st.size();

/**
* Ingresa un dato al stack
* @timecomplexity: O(1)
*/
st.push(item);
/**
* Elimina el tope del stack
* @timecomplexity: O(1)
*/
st.pop();

/**
* Retorna al elemento del tope del stack
* @timecomplexity: O(1)
*/
st.top();
```

3.3. Queue

La Queue es la estructura de datos que cumple con la propiedad de FIFO

```
/**
* Declaración de una queue del tipo T
*/
std::queue<T> queue;

/**
* Retorna el tamaño del queue
* @timecomplexity: O(1)
*/
queue.size();

/**
* Ingresa un elemento a la queue
* @timecomplexity: O(1)
*/
queue.push(item);

/**
* Elimina el elemento al inicio de la queue
* @timecomplexity: O(1)
*/
queue.pop();

/**
* Retorna el elemento al frente de la queue
* @timecomplexity: O(1)
*/
queue.top();
```

3.4. Priority Queue

Una priority queue permite mantener los datos de mayor prioridad con acceso $O(1)$. Esta mayor prioridad generalmente se basa en base al valor.

```
/**
* Crea una priority queue del tipo T
* El elemento al tope de la priority queue sera el de mayor valor
*/
std::priority_queue<T> pq;

/**
* Crea una priorty queue del tipo T
* El elemento al tope de la priority queue sera aquel de menor valor
*/
std::priority_queue<T, std::vector<T>, std::greater<T>> min_pq;

/**
* Retorna el tamaño de la priority queue
* @timecomplexity: O(1)
*/
pq.size();

/**
* Inserta un elemento en la priority queue
* @timecomplexity: O(\log{n})
*/
pq.insert(item);

/**
* Elimina el elemento de mayor prioridad
* @timecomplexity: O(\log{n})
*/
pq.pop();

/**
* Obtiene el elemento al tope de la priority queue
* @timecomplexity: O(1)
*/
pq.top();
```

3.5. Set

Un set es una estructura de datos que permite guardar elementos unicos.

```
/**
* Declaración de un set de tipo T
*/
std::set<T> set;

/**
* Retorna el tamaño del set
* @timecomplexity: O(1)
*/
set.size();

/**
* Inserta un elemento al set
* @timecomplexity: O(\log{n})
*/
set.insert(item);

/**
* Elimina el elemento del set
* @timecomplexity: O()
*/
set.erase(item);

/**
* Retorna un iterador al elemento item. En caso que no existe retorna un iterador al final
* @timecomplexity: O(\log{n})
*/
set.find(item);

/**
* Recorrer el set con un iterador. Permite obtener los elementos en orden basado en
↪  red-black-tree
* @timecomplexity: O(n)
*/
for(auto itr = set.begin(); itr != set.end(); itr++){}
```

Unordered Set

Un unordered set cumple con las mismas propiedades que el set, pero con una complejidad distinta. Su complejidad pasa a ser $O(1)$ dada a pasar de un Red-Black Tree a funciones hash.

```
/**
* Declaración de un unordered_set de tipo T
*/
std::unordered_set<T> unordered_set;

/**
* Retorna el tamaño del unordered_set
* @timecomplexity: O(1)
*/
unordered_set.size();

/**
```

```
* Inserta un elemento al unordered_set
* @timecomplexity: O(1)
*/
unordered_set.insert(item);

/**
* Elimina el elemento del unordered_set
* @timecomplexity: O(1)
*/
unordered_set.erase(item);

/**
* Retorna un iterador al elemento item. En caso que no existe retorna un iterador al final
* @timecomplexity: O(1)
*/
unordered_set.find(item);
```

3.6. Map

Un map es una estructura de datos que guarda valores en la forma llave : valor.

```
/**
* Declaración de un map con llave del tipo T1 y valor del tipo T2
*/
std::map<T1,T2> map;

/**
* Retorna el tamaño del map
* @timecomplexity: O(1)
*/
map.size();

/**
* Inserta un par llave - valor dentro del map
* @timecomplexity: O(\log{n})
*/
map.insert({llave,valor});

/**
* Elimina el par que contenga la llave
* @timecomplexity: O(\log{n})
*/
map.erase(llave);

/**
* Retorna un iterador hacia item. Retorna un iterador al final si no existe
* @timecomplexity: O(\log{n})
*
*/
itr = map.find(llave);

/**
* Acceso al primer elemento (llave)
*/
itr -> first;

/**
* Acceso al segundo elemento (valor)
*/
itr -> second;
```

Unordered Map

El unordered map cumple con las mismas propiedades que un Map. Su complejidades pasan a ser $O(1)$, dado de pasar de un Red-Black tree a un hash

```
/**
* Declaración de un unordered_map con llave del tipo T1 y valor del tipo T2
*/
std::unordered_map<T1,T2> unordered_map;

/**
* Retorna el tamaño del unordered_map
```

```
* @timecomplexity: O(1)
*/
unordered_map.size();

/**
* Inserta un par llave - valor dentro del unordered_map
* @timecomplexity: O(1)
*/
unordered_map.insert({llave,valor});

/**
* Elimina el par que contenga la llave
* @timecomplexity: O(1)
*/
unordered_map.erase(llave);

/**
* Retorna un iterador hacia item. Retorna un iterador al final si no existe
* @timecomplexity: O(1)
*
*/
itr = unordered_map.find(llave);

/**
* Acceso al primer elemento (llave)
*/
itr -> first;

/**
* Acceso al segundo elemento (valor)
*/
itr -> second;
```

3.7. Disjoint Union Set

El disjointed union set permite identificar si dos elementos estan del mismo set. Si no entiendes su funcionamiento, no pasa nada. Utiliza la siguiente implementación para algoritmos como Kruskal

```
/**
* Clase que implementa un Disjoint Set Union (DSU)
*/
class DisjointSetUnion{

private:

    std::vector<int> parent;
    std::vector<int> rank; // Para mantener la altura del arbol

public:

    /**
    *
    * @time_complexity: O(n)
    * @space_complexity: O(n)
    */
    DisjointSetUnion(int n){
        parent.resize(n);
        std::iota(parent.begin(), parent.end(), 0);
        rank.assign(n, 0);
    }

    /**
    *
    * @time_complexity: O(\alpha(N))*
    * @space_complexity: O(n)
    */
    int find(int i) {
        if (parent[i] == i) {return i; }
        return parent[i] = find(parent[i]);
    }

    /**
    *
    * @time_complexity: O(\alpha(N))*
    * @space_complexity: O(n)
    */
    bool unite(int x, int y){
        int root_x = find(x);
        int root_y = find(y);

        if(root_x != root_y){

            if (rank[root_x] < rank[root_y]) {
            parent[root_x] = root_y;
            } else if (rank[root_x] > rank[root_y]) {
                parent[root_y] = root_x;
            } else {
                parent[root_y] = root_x;
                rank[root_x]++;
            }
            return true;
```

```
            }

            return false;
        }

    };
```

4 Librería <*algorithm*> C++

apuntes

4.1. sección

apuntes

5 Algoritmos de Sorting

6 Grafos

Los grafos son estructuras de datos que permiten conectar nodos y caminos.

6.1. Implementación de Grafo

Esta implementación es utilizada en todos los problemas. Corresponde a una implementación como lista enlazada.

```
/**
 * Clase que implementa un grafo mediante lista de adyacencia
 *
 * Esta implementación tiene los siguientes supuestos
 * - El valor del nodo corresponde a su indice. (Nodo 0 == nodes[0])
 * - No habra multiples aristas en la misma dirección
 * - No es posible borrar nodos (Es programable, pero depende del problema)
 */
class GraphAdjacency{


    private:

        /**
         * @brief Representa una arista con dirección a v y peso w
         */
        struct Edge{
            int node_v; // Indice / Valor de la variable de destino
            int weight; // Peso de la arista
        };

        // Definición de estados
        const int ACTIVE = 1;
        const int INACTIVE = 0;

        // Defincion de estados para DFS
        const int NOT_VISITED = 0;
        const int IN_PROCESS = 1;
        const int VISITED = 2;

        // Definición de estados para Bellman
        const int INF = INT_MAX;
        const int MINF = INT_MIN;

        /**
         * @brief Representa un nodo con valor asociado y estado, utilizado par avarios algoritmos
         */
        struct Node{
            int node_val; // Posible valor a guardar si indice no equivale
            int state; // Estado del nodo (Utilizado para algoritmos)
```

```
        std::vector<Edge> adj; // Lista de adyacencia del nodo actual
    };

    int n_nodes; // Cantidad de nodos en el grafo
    int m_edges; // Cantidad de aristas en el grafo

    std::vector<Node> nodes; // Vector que guardas los nodos relacionados al grafo.



public:

    GraphAdjacency(){
        n_nodes = 0;
        m_edges = 0;
    }

    /**
     * Agrega un nodo al grafo
     * Por convención, este sera accesible por el indice, no su valor guardado
     *
     * @param node_val Valor para guardar dentro del nodo,
     */
    void addNode(int node_val){

        Node new_node = {node_val,ACTIVE,{}};
        nodes.push_back(new_node);
        n_nodes++;

    }

    /**
     * Agrega una arista al grafo
     * Si la arista ya existe, modificara su peso
     *
     * @param node_u Indice para acceder al nodo inicial
     * @param node_v Indice para acceder al nodo destino
     * @param weight Peso de la arista
     */
    void addEdge(int node_u, int node_v, int weight){

        if(node_u < n_nodes && node_v < n_nodes){

            std::vector<Edge>& adj = nodes[node_u].adj;

            for(Edge edge : adj){
                if(edge.node_v == node_v){
                    modifyEdge(node_u,node_v,weight);
                    return;
                }
            }

            nodes[node_u].adj.push_back({node_v,weight});
            m_edges++;
        }

    }
```

```
        /**
         * Borra una arista al grafo
         *
         * @param node_u Indice para acceder al nodo inicial
         * @param node_v Indice para acceder al nodo destino
         */
        void deleteEdge(int node_u, int node_v){

            if(node_u < n_nodes && node_v < n_nodes){

                std::vector<Edge>& adj = nodes[node_u].adj;
                auto itr = adj.begin();

                while(itr != adj.end()){
                    if(itr->node_v == node_v){
                        adj.erase(itr);
                        m_edges--;
                        break;
                    }
                    else{itr++;}
                }
            }

        }

        /**
         * Modifica el peso de una arista del grafo
         *
         * @param node_u Indice para acceder al nodo inicial
         * @param node_v Indice para acceder al nodo destino
         * @param weight Peso de la arista
         */
        void modifyEdge(int node_u, int node_v, int new_weight){

            if(node_u < n_nodes && node_v < n_nodes){
                for(auto itr = nodes[node_u].adj.begin(); itr != nodes[node_u].adj.end(); itr++){
                    if(itr->node_v == node_v){itr->weight = new_weight;}
                }
            }
        }

        /**
         * Modifica el estado de un nodo
         *
         * @param node_u Indice para acceder al nodo inicial
         * @param new_state Nuevo estado para el nodo.
         */
        void modifyNodeState(int node_u, int new_state){

            if(node_u < n_nodes){
                nodes[node_u].state = new_state;
            }
        }

        /**
         * Modifica todo los valores de los nodos al requerido
         *
         * @param new_state Nuevo estado a incluir en todos los nodos
```

```
         */
        void modifyAllNodeState(int new_state){

            for(int curr_node = 0; curr_node < n_nodes; curr_node++){

                modifyNodeState(curr_node,new_state);

            }

        }
}
```

6.2. BFS: Breadth-First Search

El algoritmo de BFS encuentra los nodos visitados.

```
/**
* Realiza el proceso de BFS en el grafo.
*
* @param start_node Nodo desde que se realiza el BFS
*
* @return No retorna nada, pero deja a los estados de los nodos como visitados o no visitados.
*
* @timecomplexity: O(V + E), con V = #vertices y E = #edges
* @spacecomplexity: O(V)
*/
void BFS(int start_node){

    modifyAllNodeState(NOT_VISITED);

    std::queue<int> queue;

    nodes[start_node].state = VISITED;
    queue.push(start_node);

    while(!queue.empty()){

        int node_u = queue.front();
        queue.pop();

        for(Edge edge : nodes[node_u].adj){

            int node_v = edge.node_v;

            if(nodes[node_v].state == NOT_VISITED){

                nodes[node_v].state = VISITED;
                queue.push(node_v);

            }

        }

    }
```

```
}
```

6.3. Minimum Spanning Tree

El algoritmo de Minimum Spanning Tree permite encontrar el arbol recubridor minimo (aquel que recorre todos los nodos).
La implementación de este algoritmo puede ser mediante Prims o Kruskal. La que se encuentra en la siguiente es Kruskal.

```
    struct KruskalEdge{
            int node_u;
            int node_v;
            int weight;
        };

        /**
         * Obtiene el Minimum Spanning Tree usando Kruskal
         *
         * @pre El grafo debe ser unidireccionado. Para esto, asegurarse de usar AddEdge en ambos
         ↪  sentidos
         * @pre Debe incluirse una implementación de Disjoint Set Union (DSU)
         * @pre Utilizar el struct definido para guardar el MSP, KruskalEdge
         *
         * @return Valor
         *
         * @timecomplexity: O(E \log{E}) o O(E \log{V})
         * @spacecomplexity: O(V + E)
         */
        std::vector<KruskalEdge> minimumSpanningTreeKruskal(){

            std::vector<KruskalEdge> all_edges;

            for (int node_u = 0; node_u < n_nodes; node_u++) {

                std::vector<Edge>& adj = nodes[node_u].adj;

                for (Edge edge : adj) {
                    int v = edge.node_v;
                    if (node_u < v) {
                        all_edges.push_back({node_u, v, edge.weight});
                    }
                }
            }

            std::sort(all_edges.begin(), all_edges.end(), [](const KruskalEdge& a, const
            ↪  KruskalEdge& b) {
                return a.weight < b.weight;
            });

            DisjointSetUnion dsu(n_nodes);
            std::vector<KruskalEdge> mst_edges;

            for (KruskalEdge edge : all_edges) {
                if (dsu.unite(edge.node_u, edge.node_v)) {
```

```
                mst_edges.push_back(edge);
            }

            if (mst_edges.size() == n_nodes - 1) {
                break;
            }
        }

        return mst_edges;
    }
```

7 Fuerza bruta

7.1. Problemas Fuerza Bruta

La siguiente sección incluye todos los problemas de fuerza bruta realizados

8 Algoritmos Greedy

Mas que otra cosa, los algoritmos greedy es como se te ocurre finalmente el ejercicio.
Intentar evitar.

9 Programación dinámica

Programación dinamica es una forma de resolver algoritmos para optimizar los recursos recursivos de un problema.
La idea principal de DP (dynamic programming) es intentar memorizar estados previos para no tener que calcularnos nuevamente en cada iteración.

9.1. Metodos de resolución de DP

Memorización

La memorización es un metodo del tipo Top Down. En esta, se tiene una EDD que mantenga guardados los datos calculados previamente mas los datos base. Si al intentar calcular un elemento este no se encuentra dentro de la memoria, entonces ahi recien calcula, para luego guardarlo en memoria.

Tabulation

Tabulación es un metodo Bottom Up, en el cual primero se calculan las instancias mas pequeñas del problema. Se utiliza una tabla dp que contiene las primeras soluciones para caso base, para luego ir rellenando con formula recursiva. La llamada recursiva es en la tabla, no en la función.

9.2. Problemas DP Lineal

Estos tipos de problemas son aquellos que tienen trancisiones consistentes. Generalmente son completables en tiempo lineal.
Las soluciones de estos problemas son usando un arreglo de prefijados, guardando aquel que responda a la necesidad del problema

Escaleras para salvar el ramo

Los estudiantes de Algoritmos y Complejidad se enfrentan a un reto tras un desafiante primer certamen. Con el objetivo de mejorar sus puntajes, exploran estrategias oscuras. Sebastián, uno de los ayudantes, les propone un desafío interesante: cada estudiante comienza con 1000 puntos y debe subir una escalera numerada en la que cada peldaño tiene una penalización en puntos. Al pisar un peldaño, el puntaje indicado en él se descuenta de su total. Los estudiantes pueden avanzar 1, 2 o 3 peldaños a la vez, permitiéndoles saltar algunos peldaños para evitar penalizaciones mayores.
El objetivo es encontrar la estrategia óptima para llegar al último peldaño, partiendo desde el primero, con el mayor puntaje posible

Input

La entrada comienza con un entero n que indica el numero de peldaños disponibles ($1 \leq n \leq 10^5$)
En las siguiente n lineas, se encuentra las penalizaciones de cada peldaño ($0 \leq p_i \leq 500$)

Output

Imprime un único entero que indica el máximo puntaje posible al llegar al último escalón.

Solución

```
ll min_pen(ll n_escaleras, std::vector<ll>&penalizaciones){

    // Dp: minima penalizacion en el escalon i
    std::vector<ll> dp_pen(n_escaleras,LLONG_MAX);

    // caso base:

    dp_pen[0] = penalizaciones[0];
    if(n_escaleras>=2) dp_pen[1] = dp_pen[0] + penalizaciones[1]; //desde 0 salte 1
    if(n_escaleras>=3) dp_pen[2] = std::min(dp_pen[0],dp_pen[1]) + penalizaciones[2]; // desde 0/1
     ↪  salte a 2

    // iteraciones

    for(ll curr_escalon = 3; curr_escalon < n_escaleras; curr_escalon++){

        dp_pen[curr_escalon] =
         ↪  std::min({dp_pen[curr_escalon-3],dp_pen[curr_escalon-2],dp_pen[curr_escalon-1]}) +
         ↪  penalizaciones[curr_escalon];

    }

    return dp_pen[n_escaleras-1];

}

void solve(){

    ll n_escaleras;
    std::cin >> n_escaleras;

    std::vector<ll> penalizaciones(n_escaleras);
    for(ll curr_p = 0; curr_p < n_escaleras; curr_p++){
        std::cin >> penalizaciones[curr_p];
    }

    std::cout << 1000-min_pen(n_escaleras,penalizaciones) << std::endl;

}

int main(){
    optimize()
    solve();
    return 0;
}
```

House Robber

You are a professional robber planning to rob houses along a street. Each house has a certain amount of money stashed, the only constraint stopping you from robbing each of them is that adjacent houses have security systems connected and it will automatically contact the police if two adjacent houses were broken into on the same night. Given an integer array nums representing the amount of money of each house, return the maximum amount of money you can rob tonight without alerting the police.

```cpp
class Solution {
public:
    int rob(vector<int>& nums) {

        int n_casas = nums.size();

        // casos basicos que no necesitan dp
        if(nums.size() == 1){
            return nums[0];
        }

        if(nums.size() == 2){
            return std::max(nums[0], nums[1]);
        }

        // dp
        std::vector<int> dp_value(n_casas, 0);

        // caso base: El maximo valor que puedo robar en la primera casa es su propio valor
        // Lo mismo para la segunda casa, solo que en esta no se incluye la primera

        dp_value[0] = nums[0];
        dp_value[1] = std::max(nums[1],nums[0]);

        // iteraciones

        for(int curr_idx = 2; curr_idx < n_casas; curr_idx++){

            int curr_house_value = nums[curr_idx];

            dp_value[curr_idx] = std::max(dp_value[curr_idx-1], dp_value[curr_idx-2] +
            ↪  curr_house_value);

        }

        return dp_value[n_casas-1];

    }
};
```

9.3. Problemas DP Secuenciales

Estos problemas generalmente se escriben de la forma dp[i][j], donde se tiene un valor de una secuencia 1 con largo i, mientras se tiene una secuencia 2 con largo j

Longest Common Subsequence

Problem Statement
You are given strings s and t. Find one longest string that is a subsequence of both s and t. Notes
A subsequence of a string x is the string obtained by removing zero or more characters from x and concatenating the remaining characters without changing the order.

Input

Input is given from Standard Input in the following format:

s
t

Output

Print one longest string that is a subsequence of both s and t. If there are multiple such strings, any of them will be accepted.

Example 1

axyb
abyxb
Result: axb or ayb

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

#define optimize() std::ios_base::sync_with_stdio(false) ; std::cin.tie(NULL) ;
↪  std::cout.tie(NULL) ;

#define ll long long

std::string reconstruct(std::string &S_string, std::string &T_string,
↪  std::vector<std::vector<ll>>& dp){

    ll i = S_string.size();
    ll j = T_string.size();
    std::string result = "";

    while(i > 0 && j > 0){
        if(S_string[i-1] == T_string[j-1]){
            result.push_back(S_string[i-1]);
            i--;
            j--;
        }
        else if(dp[i-1][j] >= dp[i][j-1]){ i--; }
        else{ j--; }

    }

    std::reverse(result.begin(), result.end());
    return result;

}

void tabulation(std::string &S_string, std::string &T_string, std::vector<std::vector<ll>>& dp){

    // Ignorable, dado que dp inicio con valores 0.
    dp[S_string.size()][0] = 0;
    dp[0][T_string.size()] = 0;

    for(ll i=1; i <= S_string.size(); i++){
        for(ll j=1; j <= T_string.size(); j++){

            if(S_string[i-1] == T_string[j-1]){ dp[i][j] = dp[i-1][j-1] + 1; }
            else{ dp[i][j] = std::max( dp[i-1][j], dp[i][j-1]); }
        }
    }
}
```

```
void testcase(){

    std::string S_string;
    std::string T_string;

    std::cin >> S_string >> T_string;

    std::vector<std::vector<ll>> dp(S_string.size()+1, std::vector<ll>(T_string.size()+1,0));
    tabulation(S_string,T_string,dp);
    std::cout << reconstruct(S_string,T_string,dp);

}

int main(){
    testcase();
    return 0;
}
```

9.4. Problemas DP Knapsack

El problema de Knapsack se tiene de dos formatos distintos:

- **Knapsack 0/1**: Estos problemas tienen el requerimiento de que un item se puede tomar unicamente una vez. Generalmente siguen el formato dp[i][w], donde representa el valor buscado tomando i cantidad de items, y con peso w
- **Unbonded Knapsack**: Estos problemas pueden repetir un item cuantas veces quieran. Generalmente son de la forma dp_value[cant]

Knapsack 0/1

Implement a solution to the classic knapsack problem. You are given a knapsack that can hold up to a certain weight (its capacity), and several items you may choose to put in the knapsack. Each item has a weight and a value. Choose a subset of the items (which could be all of them, or none of them) having the greatest value that fit into the knapsack (i.e. the sum of the weights of the items you choose must be less than or equal to the knapsack capacity).

Input

The input consists of between 1 and 30 test cases. Each test case begins with an integer $1 \leq C \leq 2000$, giving the capacity of the knapsack, and an integer $1 \leq n \leq 2000$, giving the number of objects. Then follow n lines, each giving the value and weight of the n objects. Both values and weights are integers between 1 and 10000.

Output

For each test case, output a line containing the number of items chosen, followed by a line containing the indices of the chosen items (the first item has index 0, the second index 1, and so on). The indices can be given in any order.

Example 1:

5 3
1 5
10 5
100 5 6 4
5 4

4 3
3 2
2 1

Output Example 1:

1
2
3
1 2 3

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

#define ll long long

struct Object{
    ll value;
    ll weight;
};

void knapstack(ll capacity, std::vector<Object>& objects, std::vector<std::vector<ll>>& dp){

    for(ll i=1; i<=objects.size(); i++){
        ll curr_value = objects[i-1].value;
        ll curr_weight = objects[i-1].weight;

        for(ll w=1; w<=capacity; w++){
            if(curr_weight > w){ dp[i][w] = dp[i-1][w]; }
            else{
                ll not_take_value = dp[i-1][w];
                ll take_value = curr_value + dp[i-1][w - curr_weight];
                dp[i][w] = std::max(not_take_value, take_value);
            }
        }

    }

    std::vector<ll> selected_objects;
    ll curr_weight = capacity;

    for(ll i = objects.size(); i > 0; i--){

        if(dp[i][curr_weight] != dp[i-1][curr_weight]){
            selected_objects.push_back(i-1);
            curr_weight -= objects[i-1].weight;
        }
    }

    // Output

    std::cout << selected_objects.size() << std::endl;
    for(ll i=0; i<selected_objects.size(); i++){
        std::string s = " ";
        if(i == selected_objects.size()-1){ s = ""; }
        std::cout << selected_objects[i] << s;
    }
    std::cout << std::endl;
```

```
}

int main(){

    ll capacity, n_objects;
    while(std::cin >> capacity >> n_objects){
        std::vector<Object> objects(n_objects);
        for(ll i=0; i<n_objects; i++) { std::cin >> objects[i].value >> objects[i].weight; }

        std::vector<std::vector<ll>> dp(n_objects+1, std::vector<ll>(capacity+1, 0));
        knapstack(capacity,objects,dp);
    }

    return 0;
}
```

Vacaciones

Taro's summer vacation starts tomorrow, and he has decided to make plans for it now.
The vacation consists of N days. For each i ($1 \leq i \leq N$), Taro will choose one of the following activities and do it on the i-th day:

- **A**: Swim in the sea. Gain a_i points of happiness.
- **B**: Catch bugs in the mountains. Gain b_i points of happiness.
- **C**: Do homework at home. Gain c_i points of happiness.

As Taro gets bored easily, he cannot do the same activities for two or more consecutive days.
Find the maximum possible total points of happiness that Taro gains.

Constraints

- All values in input are integers.
- $1 \leq N \leq 10^5$
- $1 \leq a_i, b_i, c_i \leq 10^4$

Input Example 1

3
10 40 70
20 50 80
30 60 90

Output Example 1

210

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

#define optimize() std::ios_base::sync_with_stdio(false) ; std::cin.tie(NULL) ;
↪  std::cout.tie(NULL) ;

#define ll long long
```

```cpp
#define A_INDEX 0
#define B_INDEX 1
#define C_INDEX 2

struct DayActivity{
    ll A;
    ll B;
    ll C;
};

ll tabulation(std::vector<DayActivity> &activities, std::vector<std::vector<ll>> &dp){

    dp[0][A_INDEX] = 0;
    dp[0][B_INDEX] = 0;
    dp[0][C_INDEX] = 0;

    for(ll i=1; i<=activities.size(); i++){

        dp[i][A_INDEX] = activities[i-1].A + std::max(dp[i-1][B_INDEX], dp[i-1][C_INDEX]);
        dp[i][B_INDEX] = activities[i-1].B + std::max(dp[i-1][A_INDEX], dp[i-1][C_INDEX]);
        dp[i][C_INDEX] = activities[i-1].C + std::max(dp[i-1][A_INDEX], dp[i-1][B_INDEX]);

    }

    return
    ↪  std::max({dp[activities.size()][A_INDEX],dp[activities.size()][B_INDEX],dp[activities.size()][C_INDEX]});

}

void testcase(){

    ll N_DAYS;
    std::cin >> N_DAYS;

    std::vector<DayActivity> activities(N_DAYS);
    for(ll i=0; i<N_DAYS; i++){std::cin >> activities[i].A >> activities[i].B >> activities[i].C;
    ↪  }

    std::vector<std::vector<ll>> dp(N_DAYS+1, std::vector<ll>(3,0));
    std::cout << tabulation(activities,dp) << std::endl;

}

int main(){
    optimize();
    testcase();
    return 0;
}
```

9.5. Problemas DP de Intervalos

En estos tipos de problemas, el problema es resuelto en cada intervalo (subarreglo) del arreglo.

Increasing Subsequence

A **strictly increasing sequence** is a sequence of numbers $a_1, a_2, \ldots, a_n$ such that, for $1 < i \leq n$, $a_{i-1} < a_i$. A subsequence of $a_1, a_2, \ldots, a_n$ is identified by a strictly increasing sequence of indices, $x_1, x_2, \ldots, x_m$ where $1 \leq x_1$ and $x_m \leq n$. We say $a_{x_1}, a_{x_2}, \ldots, a_{x_m}$ is a subsequence of $a_1, a_2, \ldots, a_n$. For example, given the sequence $8, 90, 4, 10\,000, 2, 18, 60, 172, 99$, we can say that $90, 4, 10\,000, 18$ is a subsequence but $8, 90, 18, 2, 60$ is not. The subsequence $4, 18, 60, 172$ is a subsequence that is, itself, strictly increasing.
Given a sequence of numbers, can you write a program to find a strictly increasing subsequence that is as long as possible?

Input

Input has up to 200 test cases, one per line. Each test case starts with an integer $1 \leq n \leq 200$, followed by n integer values, all in the range $[0, 10^8]$. A value of zero for n marks the end of input.

Output

For each test case, output the length of the longest strictly increasing subsequence, followed by the values of the **lexicographically-earliest** such sequence. A sequence $a_1, a_2, \ldots, a_m$ is lexicographically earlier than $b_1, b_2, \ldots, b_m$ if some $a_i < b_i$ and $a_j = b_j$ for all $j < i$.

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

#define optimize() std::ios_base::sync_with_stdio(false) ; std::cin.tie(NULL) ;
↪  std::cout.tie(NULL) ;

#define ll long long

std::vector<ll> reconstruct(std::vector<ll> &values, std::vector<ll> &dp_last_item, ll last_idx){

    std::stack<ll> stack;

    while(last_idx != -1){

            stack.push(values[last_idx]);
            last_idx = dp_last_item[last_idx];

        }

        std::vector<ll> result;
        while(!stack.empty()){
            result.push_back(stack.top()); stack.pop();
        }

    return result;

}

void longestIncreasingSubsequence(std::vector<ll> &values, std::vector<ll> &dp_subsequence,
↪  std::vector<ll> &dp_last_item){

    ll n = values.size();

    for(ll curr_idx = 1; curr_idx < n; curr_idx++){
        for(ll prev_idx = 0; prev_idx < curr_idx; prev_idx++){

            if(values[curr_idx] > values[prev_idx]){
```

```
                if(dp_subsequence[curr_idx] < dp_subsequence[prev_idx] + 1){
                    dp_subsequence[curr_idx] = dp_subsequence[prev_idx] + 1;
                    dp_last_item[curr_idx] = prev_idx;
                }
                else if(dp_subsequence[curr_idx] == dp_subsequence[prev_idx] + 1){

                    std::vector<ll> s1 = reconstruct(values,dp_last_item,curr_idx);
                    std::vector<ll> s2 = reconstruct(values,dp_last_item,prev_idx);
                    s2.push_back(values[curr_idx]);

                    //std::cout << "s1: "; printArray(s1);
                    //std::cout << "s2: "; printArray(s2);

                    if(s2 < s1){
                        dp_last_item[curr_idx] = prev_idx;
                        //std::cout << "winner: s1" << std::endl;
                    }

                }

            }

        }
    }
}

void solve(){

    ll n_values;
    while(std::cin >> n_values && n_values != 0){

        std::vector<ll> values(n_values);
        for(ll curr_value_idx = 0; curr_value_idx < n_values; curr_value_idx++){
            std::cin >> values[curr_value_idx];

        }

        // dp que mantiene guardado el valor de la subsequence mas grande en el index i
        std::vector<ll> dp_subsequence(values.size(), 1);
        // dp que mantiene el ultimo item para forma la subsequencia en el index i
        std::vector<ll> dp_last_item(values.size(),-1);

        longestIncreasingSubsequence(values, dp_subsequence, dp_last_item);

        ll max_len = 0;
        for(ll curr_idx = 0; curr_idx < values.size(); curr_idx++){
            if( dp_subsequence[curr_idx] > max_len){max_len = dp_subsequence[curr_idx];}
        }

        std::vector<ll> LIS;
        bool first = true;

        for(ll curr_idx = 0; curr_idx < values.size(); curr_idx++){
            if(dp_subsequence[curr_idx] == max_len){
                std::vector<ll> curr_LIS = reconstruct(values,dp_last_item,curr_idx);
                if(first || curr_LIS < LIS ){
                    LIS = curr_LIS;
```

```
                    first = false;
                }
            }
        }

        std::cout << LIS.size() << " ";
        for(ll curr_idx = 0; curr_idx < LIS.size(); curr_idx++){
            std::cout << LIS[curr_idx];
            if(curr_idx != LIS.size()-1){std::cout << " ";}
        }
        std::cout << std::endl;

    }

}

int main(){
    optimize();
    solve();
    return 0;
}
```

9.6. Problemas DP Bellman

El problema de encontrar caminos minimos con ciclos negativos es resuelto por Bellman.

Single source shortest path, negative weights (Bellmans)

Input

The input consists of several test cases. Each test case starts with a line with four non-negative integers, $1 \leq n \leq 1000$, $0 \leq m \leq 5000$, $1 \leq q \leq 100$ and $0 \leq s < n$, separated by single spaces, where n is the number of nodes in the graph, m the number of edges, q the number of queries and s the index of the starting node. Nodes are numbered from 0 to $n-1$. Then follow m lines, each line consisting of three (space-separated) integers u, v and w indicating that there is an edge from u to v in the graph with weight $-2000 \leq w \leq 2000$. Then follow q lines of queries, each consisting of a single non-negative integer, asking for the minimum distance from node s to the node number given on the query line.
Input will be terminated by a line containing four zeros, this line should **not** be processed.

Output

For each query, output a single line containing the minimum distance from node s to the node specified in the query, the word Ïmpossible" if there is no path from s to that node, or Infinity" if there are arbitrarily short paths from s to that node. For clarity, the sample output has a blank line between the output for different cases.

```
/**
    * Realiza el algoritmo de Bellman para encontrar distancias con aristas de peso negativo
    * Esta implementación considera el Grafo enlazado en partes previas del documento. Utilizar
    ↪  aquel.
    *
    * @param s_node Nodo incial donde realizar el Bellman
    * @param ref_node Nodo que se quiere saber su valor de distancia desde s_node
    *
    * @return Valor
    *
```

```
    * @timecomplexity: O(V * E)
    * @spacecomplexity: O(2V) = O(V)
    */
int bellman(int s_node, int ref_node){

    std::vector<long long> dp_distances(n_nodes, INF); // Utiliza long long por si hay overflow
    std::vector<bool> dp_negative_cycle(n_nodes, false);

    // Caso base: dp[s_node]

    dp_distances[s_node] = 0;

    // Relajacion: n-1 iteraciones

    for(int curr_itr = 0; curr_itr < n_nodes - 1; curr_itr++){
        for(int curr_node = 0; curr_node < n_nodes; curr_node++){

            if(dp_distances[curr_node] == INF){continue;}

            std::vector<Edge>& adj = nodes[curr_node].adj;

            for(Edge edge : adj){

                int node_v = edge.node_v;

                long long new_dist = dp_distances[curr_node] + edge.weight;

                if (new_dist < dp_distances[node_v]){
                    dp_distances[node_v] = new_dist;
                }
            }
        }
    }

    // Detección y Propagación de Ciclos Negativos

    // Primero, detectar qué nodos pueden ser relajados en la n-ésima iteración
    for(int curr_node = 0; curr_node < n_nodes; curr_node++){

        if(dp_distances[curr_node] == INF){ continue; }

        std::vector<Edge>& adj = nodes[curr_node].adj;

        for(Edge edge : adj){

            int node_v = edge.node_v;
            long long new_dist = dp_distances[curr_node] + edge.weight;

            // Si se puede relajar en la n-ésima iteración, está en un ciclo negativo o es
            ↪  alcanzable desde uno.
            if(new_dist < dp_distances[node_v]){
                dp_negative_cycle[node_v] = true;
            }
        }
    }

    // Segundo, propagar el estado de ciclo negativo a todos los nodos alcanzables.
    // Esto requiere n-1 iteraciones adicionales (similar a un BFS/DFS implícito)
```

```
    for(int curr_itr = 0; curr_itr < n_nodes - 1; curr_itr++){
        for(int curr_node = 0; curr_node < n_nodes; curr_node++){

            if(dp_negative_cycle[curr_node]){

                std::vector<Edge>& adj = nodes[curr_node].adj;

                for(Edge edge : adj){
                    int node_v = edge.node_v;
                    dp_negative_cycle[node_v] = true;
                }
            }
        }
    }

    // Retorno

    if(dp_distances[ref_node] == INF){return INF;} // No alcanzable
    else if(dp_negative_cycle[ref_node] == true){return -1; } // Negative cycle
    else{return dp_distances[ref_node]; } // Resolución
}
```