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;
        }

    };
```

3.8. Grafo (Implementación con lista de adyaciencia)

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

4 Librería <*algorithm*> C++

apuntes

4.1. sección

apuntes

5 Algoritmos de Sorting

6 Grafos

7 Fuerza bruta

8 Algoritmos Greedy

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. Fibonnaci

The Fibonacci numbers, commonly denoted F(n) form a sequence, called the Fibonacci sequence, such that each number is the sum of the two preceding ones, starting from 0 and 1. That is,

$$F(0) = 0, F(1) = 1$$

$$F(n) = F(n-1) + F(n-2), \text{for} n > 1.$$

Given n, calculate F(n).
Example 1:
Input: n = 2
Output: 1
Explanation: $F(2) = F(1) + F(0) = 1 + 0 = 1$.
Example 2:
Input: n = 3
Output: 2
Explanation: $F(3) = F(2) + F(1) = 1 + 1 = 2$.
Example 3:
Input: n = 4
Output: 3
Explanation: $F(4) = F(3) + F(2) = 2 + 1 = 3$.
Constraints:
$0 <= n <= 30$

```
class Solution \{
    public:
        std::map<int, int> dp;

        Solution()\{
            dp[0] = 0;
            dp[1] = 1;
        \}

        /**
        * Calculates fibonacci sequence using a Top-Down approach
        * - Base case: 0 and 1 results added previously on memory. Memory[0] = 0 and Memory[1]
        ↪  = 1
        *
        * @param number Number of the fibonacci sequence to be calculated.
        * @param memory Map that stores previos calculated numbers in fibonnaci
        *
        */
        int fib(int n) \{

            if(dp.find(n) == dp.end()){ dp[n] = fib(n-1) + fib(n-2);}
            return dp[n];

        \}

    \};
```