# 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

## 7.1. Problemas Fuerza Bruta

La siguiente sección incluye todos los problemas de fuerza bruta realizados

# 8 Algoritmos Greedy

## 8.1. Problemas Greedy

La siguiente sección incluye todos los problemas Greedy realizados.

### Frosh Week

Professor Zac is trying to finish a collection of tasks during the first week at the start of the term. He knows precisely how long each task will take, down to the millisecond. Unfortunately, it is also Frosh Week. Zac's office window has a clear view of the stage where loud music is played. He cannot focus on any task when music is blaring.

The event organizers are also very precise. They supply Zac with intervals of time when music will not be playing. These intervals are specified by their start and end times down to the millisecond.

Each task that Zac completes must be completed in one quiet interval. He cannot pause working on a task when music plays (he loses his train of thought). Interestingly, the lengths of the tasks and quiet intervals are such that it is impossible to finish more than one task per quiet interval!

Given a list of times $t_i$ (in milliseconds) that each task will take and a list of times $\ell_j$ (in milliseconds) specifying the lengths of the intervals when no music is being played, what is the maximum number of tasks that Zac can complete?

#### Input

The first line of input contains a pair of integers $n$ and $m$, where $n$ is the number of tasks and $m$ is the number of time intervals when no music is played. The second line consists of a list of integers $t_1, t_2, \ldots, t_n$ indicating the length of time of each task. The final line consists of a list of times $\ell_1, \ell_2, \ldots, \ell_m$ indicating the length of time of each quiet interval when Zac is at work this week.

You may assume that $1 \leq n, m \leq 200\,000$ and $100\,000 \leq t_i, \ell_j \leq 199\,999$ for each task $i$ and each quiet interval $j$.

#### Output

Output consists of a single line containing a single integer indicating the number of tasks that Zac can accomplish from his list during this first week.

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

#define ll long long

int main(){

    ll t;
    std::cin >> t;

    while(t--){

        ll l, n;
        std::cin >> l >> n;

        ll input;
        std::vector<ll> positions;

        for(int i=0; i<n; i++){
            std::cin >> input;
            positions.push_back(input);
        }

        ll min = 0;
        ll max = 0;

        for(int i=0; i<n; i++){

            ll curr_pos = positions[i];
```

```
            ll left_distance = curr_pos;
            ll right_distance = l - curr_pos;

            ll min_curr_distance = std::min(left_distance,right_distance);
            min = std::max(min_curr_distance,min);

            ll max_curr_distance = std::max(left_distance,right_distance);
            max = std::max(max_curr_distance,max);
        }

        std::cout << min << " " << max << std::endl;

    }

    return 0;
}
```

### Assigning Workstations

Penelope is part of the admin team of the newly built supercomputer. Her job is to assign workstations to the researchers who come here to run their computations at the supercomputer.
Penelope is very lazy and hates unlocking machines for the arriving researchers. She can unlock the machines remotely from her desk, but does not feel that this menial task matches her qualifications. Should she decide to ignore the security guidelines she could simply ask the researchers not to lock their workstations when they leave, and then assign new researchers to workstations that are not used any more but that are still unlocked. That way, she only needs to unlock each workstation for the first researcher using it, which would be a huge improvement for Penelope.
Unfortunately, unused workstations lock themselves automatically if they are unused for more than $m$ minutes. After a workstation has locked itself, Penelope has to unlock it again for the next researcher using it. Given the exact schedule of arriving and leaving researchers, can you tell Penelope how many unlockings she may save by asking the researchers not to lock their workstations when they leave and assigning arriving researchers to workstations in an optimal way?
You may assume that there are enough workstations available.

#### Input

The input consists of: * one line with two integers $n$ ($1 \leq n \leq 300\,000$), the number of researchers, and $m$ ($1 \leq m \leq 10^8$), the number of minutes of inactivity after which a workstation locks itself; * $n$ lines each with two integers $a$ and $s$ ($1 \leq a, s \leq 10^8$), representing a researcher that arrives at $a$ minutes and stays for $s$ minutes.

#### Output

Output the maximum number of unlockings Penelope may save herself.

#### Sample 1

3 5
1 5
6 3
14 6

#### Output 1:

2

#### Input

2
5 10
2 6
1 2
17 7
3 9
15 6

#### Output 2

3

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

#define ll long long
```

```
struct Worker{
    ll arrive_time;
    ll duration;
};

struct Workstation{
    ll start;
    ll end;
};

int main(){

    ll n, m;
    std::cin >> n >> m;

    std::priority_queue<ll> inactive_time;
    std::vector<Worker> workers;

    for(int i=0; i<n; i++){

        ll arrive_time, duration;
        std::cin >> arrive_time >> duration;

        Worker worker;
        worker.arrive_time = arrive_time;
        worker.duration = duration;

        workers.push_back(worker);

    }

    std::sort(workers.begin(), workers.end(), [](const Worker&a, const Worker&b){
        return a.arrive_time < b.arrive_time;
    });

    ll unlockings_saved = 0;

    for(int i=0; i<n; i++){

        ll curr_workstation;
        Worker curr_worker = workers[i];

        while(!inactive_time.empty()){

            ll curr_workstation_inactive = -inactive_time.top();

            // no activity has started the inactive period
            if(curr_workstation_inactive > curr_worker.arrive_time){
                break;
            }

            // In the inactive period
            else if(curr_workstation_inactive <= curr_worker.arrive_time &&
            ↪  curr_worker.arrive_time <= curr_workstation_inactive+m){
                unlockings_saved++;
                inactive_time.pop();
                break;
```

```
            }

            // inactive_time already ended
            else{
                inactive_time.pop();
            }

        }

        curr_workstation = curr_worker.arrive_time + curr_worker.duration;
        inactive_time.push(-curr_workstation);



    }

    std::cout << unlockings_saved << std::endl;

    return 0;
}
```

### Birds on a wire

There is a long electrical wire of length $l$ centimetres between two poles where birds like to sit. After a long day at work you like to watch the birds on the wire from your balcony. Some time ago you noticed that they don't like to sit closer than $d$ centimetres from each other. In addition, they cannot sit closer than 6 centimetres to any of the poles, since there are spikes attached to the pole to keep it clean from faeces that would otherwise damage and weaken it. You start wondering how many more birds can possibly sit on the wire.
Task
Given numbers $l$ and $d$, how many additional birds can sit on the wire given the positions of the birds already on the wire? For the purposes of this problem we assume that the birds have zero width.

#### Input

The first line contains three space separated integers: the length of the wire $l$, distance $d$ and number of birds $n$ already sitting on the wire. The next $n$ lines contain the positions of the birds in any order. All number are integers, $1 \le l, d \le 1\,000\,000\,000$ and $0 \le n \le 20\,000$. (If you have objections to the physical plausibility of fitting that many birds on a line hanging between two poles, you may either imagine that the height of the line is 0 cm above ground level, or that the birds are ants instead.) You can assume that the birds already sitting on the wire are at least 6 cm from the poles and at least $d$ centimetres apart from each other.

#### Output

Output one line with one integer – the maximal number of additional birds that can possibly sit on the wire.

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

#define ll long long

int main(){

    ll l,d,n;
    std::cin >> l >> d >> n;

    std::vector<ll> positions;
    ll input;
    for(int i=0; i<n; i++){
        std::cin >> input;
        positions.push_back(input);
    }

    if(n>0){std::sort(positions.begin(), positions.end());}

    ll new_birds = 0;

    // case birds
    for(ll i=0; i<n-1; i++){

        ll pos1 = positions[i];
        ll pos2 = positions[i+1];

        new_birds += std::max((pos2-pos1)/d - 1, 0ll);
    }

    // edge cases
    if(n > 0){
        new_birds += (positions[0]-6)/d;
        new_birds += (l-6-positions[n-1])/d;
```

```
    }
    else if(l >= 12) // no birds but string
        new_birds += (l-12)/d + 1;

    std::cout << new_birds << std::endl;
    return 0;
}
```

### Shopaholic

Lindsay is a shopaholic. Whenever there is a discount of the kind where you can buy three items and only pay for two, she goes completely mad and feels a need to buy all items in the store. You have given up on curing her for this disease, but try to limit its effect on her wallet.
You have realized that the stores coming with these offers are quite selective when it comes to which items you get for free; it is always the cheapest ones. As an example, when your friend comes to the counter with seven items, costing 400400, 350350, 300300, 250250, 200200, 150150, and 100100 dollars, she will have to pay 15001500 dollars. In this case she got a discount of 250250 dollars. You realize that if she goes to the counter three times, she might get a bigger discount. E.g. if she goes with the items that costs 400400, 300300 and 250250, she will get a discount of 250250 the first round. The next round she brings the item that costs 150150 giving no extra discount, but the third round she takes the last items that costs 350350, 200200 and 100100 giving a discount of an additional 100100 dollars, adding up to a total discount of 350350.
Your job is to find the maximum discount Lindsay can get.

#### Input

The first line of input gives the number of items Lindsay is buying, $1 \leq n \leq 200000$. The next line gives the prices of these items, which are integers $1 \leq p_i \leq 200000$.

#### Output

Output one line giving the maximum discount Lindsay can get by selectively choosing which items she brings to the counter at the same time.

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

#define ll long long

int main(){

    ll n;
    std::cin >> n;

    std::priority_queue<ll> prices;

    while(n--){
        ll item_price;
        std::cin >> item_price;
        prices.push(item_price);
    }

    std::vector<ll> trio;
    ll discounts = 0;

    while(!prices.empty()){

        trio.push_back(prices.top());
        prices.pop();

        if(trio.size() == 3){
            discounts += *std::min_element(trio.begin(), trio.end());
            trio.clear();
        }

    }
```

```
    std::cout << discounts;

    return 0;
}
```

### Boiling Vegetables

The trick to boiling vegetables is to make sure all pieces are about the same size. If they are not, the small ones get too soft or the large ones get undercooked (or both). Fortunately, you have heard of the kitchen knife, but your parents' warnings of using sharp instruments still echoes in your head. Therefore you better use it as little as possible. You can take a piece of a vegetable of weight $w$ and cut it arbitrarily in two pieces of weight $w_{\text{left}}$ and $w_{\text{right}}$, where $w_{\text{left}} + w_{\text{right}} = w$. This operation constitutes a "cut". Given a set of pieces of vegetables, determine the minimum number of cuts needed to make the ratio between the smallest and the largest resulting piece go above a given threshold.

#### Input

The input starts with a floating point number $T$ with 2 decimal digits, $0{,}5 < T < 1$, and a positive integer $N \leq 1000$. Next follow $N$ positive integer weights $w_1, w_2, \ldots, w_N$. All weights are less than $10^6$.

#### Output

Output the minimum number of cuts needed to make the ratio between the resulting minimum weight piece and the resulting maximum weight piece be above $T$. You may assume that the number of cuts needed is less than 500. To avoid issues with floating point numbers, you can assume that the optimal answer for ratio $T$ is the same as for ratio $T' + 0{,}0001$.

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

#define ld float
#define ll long long

using namespace std;

int main(){

    ld T;
    cin >> T;

    ll N;
    cin >> N;

    auto comp = [](const pair<ll,ll> &a, const pair<ll,ll> &b){
        return (a.first/a.second) < (b.first/b.second);
    };

    priority_queue<pair<ll,ll>, vector<pair<ll,ll>>, decltype(comp)> ratios(comp);

    ld min_value = LONG_LONG_MAX;
    ll weight;

    while(N--){
        cin >> weight;
        ratios.push({weight, 1});
        min_value = std::min(min_value, (ld)weight);
    }

    ll cuts = 0;

    while(!ratios.empty()){
        pair<ll, ll> max_ratio = ratios.top();
        if(min_value/(max_ratio.first/max_ratio.second) >= T) break;
```

```
        ratios.pop();
        ratios.push({max_ratio.first, (max_ratio.second + 1)});
        cuts++;

        min_value = std::min(min_value, (ld)(max_ratio.first/(max_ratio.second + 1)));

    }

    std::cout << cuts;

    return 0;
}
```

### Square Peg in a Round Hole

Mr. Johnson likes to build houses. In fact, he likes it so much that he has built a lot of houses that he has not yet placed on plots. He has recently acquired $N$ circular plots. The city government has decided that there can be only one house on each plot, and a house cannot touch the boundary of the plot.
Mr. Johnson has $M$ circular houses and $K$ square houses. Help him figure out how many of the plots he can fill with houses so that he can get some money back on his investments.

#### Input

The first line of input consists of 3 space-separated integers $N$, $M$, and $K$. The second line contains $N$ space-separated integers, where the $i^{\text{th}}$ integer denotes the radius $r_i'$ of the $i^{\text{th}}$ plot. The third line contains $M$ space-separated integers, where the $i^{\text{th}}$ integer denotes the radius $r_i$ of the $i^{\text{th}}$ circular house. The fourth line contains $K$ space-separated integers, where the $i^{\text{th}}$ integer denotes the side length $s_i$ of the $i^{\text{th}}$ square house.

#### Output

Output the largest number of plots he can fill with houses.

#### Limits

- $1 \leq N, M, K, r_i, s_i \leq 100$

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

#define ld long double

int main(){

    ld n,m,k;
    std::cin >> n >> m >> k;

    std::priority_queue<ld> plot_radius;
    std::priority_queue<ld> house_radius;

    ld input;
    while(n--){
        std::cin >> input;
        plot_radius.push(input);
    }

    while(m--){
        std::cin >> input;
        house_radius.push(input);
    }

    while(k--){
        std::cin >> input;
        house_radius.push(input / std::sqrt(2.0));
    }

    long long solution = 0;

    while(!plot_radius.empty() && !house_radius.empty()){

        ld curr_plot_radius = plot_radius.top();
```

```
        plot_radius.pop();

        while(!house_radius.empty()){

            ld curr_house_radius = house_radius.top();
            house_radius.pop();
            if(curr_house_radius < curr_plot_radius){
                solution++;
                break;
            }
        }
    }

    std::cout << solution;


    return 0;
}
```

### Ants

An army of ants walk on a horizontal pole of length $l$ cm, each with a constant speed of 1 cm/s. When a walking ant reaches an end of the pole, it immediately falls off it. When two ants meet they turn back and start walking in opposite directions. We know the original positions of ants on the pole, unfortunately, we do not know the directions in which the ants are walking. Your task is to compute the earliest and the latest possible times needed for all ants to fall off the pole.

#### Input

The first line of input contains one integer giving the number of cases that follow, at most 100. The data for each case start with two integer numbers: the length $l$ of the pole (in cm) and $n$, the number of ants residing on the pole. These two numbers are followed by $n$ integers giving the position of each ant on the pole as the distance measured from the left end of the pole, in no particular order. All input integers are between 0 and $1,000,000$ and they are separated by whitespace.

#### Output

For each case of input, output two numbers separated by a single space. The first number is the **earliest** possible time when all ants fall off the pole (if the directions of their walks are chosen appropriately) and the second number is the **latest** possible such time.

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

#define ll long long

int main(){

    ll t;
    std::cin >> t;

    while(t--){

        ll l, n;
        std::cin >> l >> n;

        ll input;
        std::vector<ll> positions;

        for(int i=0; i<n; i++){
            std::cin >> input;
            positions.push_back(input);
        }

        ll min = 0;
        ll max = 0;

        for(int i=0; i<n; i++){

            ll curr_pos = positions[i];

            ll left_distance = curr_pos;
            ll right_distance = l - curr_pos;

            ll min_curr_distance = std::min(left_distance,right_distance);
            min = std::max(min_curr_distance,min);
```

```
            ll max_curr_distance = std::max(left_distance,right_distance);
            max = std::max(max_curr_distance,max);
        }

        std::cout << min << " " << max << std::endl;

    }

    return 0;
}
```

# 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

### 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];

        \}

    \};
```

### Knapsack

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){

    // Tablutaion approach to calculate values

    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]; } // if can't take
            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);
            }
        }

    }

    // Reconstructing the best set

    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]){ // If it changed, an item was included
            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;
}
```

### Restaurant Orders

A friend of yours who is working as a waiter has a problem. A group of xkcd-fans have started to come to the restaurant and order food as in the comic strip below. Each order takes him a lot of time to figure out, but maybe you can help him.

#### Task

You are to write a program that finds out what was ordered given the total cost of the order and the cost of each item on the menu.

#### Input

The input starts with a line containing one integer $n$ ($1 \leq n \leq 100$), the number of items on the menu. The next line contains $n$ space-separated positive integers $c_1, c_2, \ldots, c_n$, denoting the cost of each item on the menu in Swedish kronor. No item costs more than 1 000 SEK.
This is followed by a line containing $m$ ($1 \leq m \leq 1\,000$), the number of orders placed, and a line with $m$ orders. Each order is given as an integer $s$ ($1 \leq s \leq 30\,000$), the total cost of all ordered items in SEK.

#### Output

For each order in the input output one line as follows. If there is one **unique** order giving the specified total cost, output a space-separated list of the numbers of the items on that order in ascending order. If the order contains more than one of the same item, print the corresponding number the appropriate number of times. The first item on the menu has number 1, the second 2, and so on.
If there doesn't exist an order that gives the specified sum, output `Impossible`. If there are more than one order that gives the specified sum, output `Ambiguous`.

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

#define ll long long

#define IMPOSSIBLE 0
#define UNIQUE 1
#define AMBIGOUS 2


std::string reconstruction(std::vector<ll> &items_costs, std::vector<ll> &dp_reconstruction, ll
↪  curr_order){

    ll curr_cost = curr_order;
    std::vector<ll> solution_items;


    while(curr_cost > 0) {

        ll curr_item = dp_reconstruction[curr_cost];

        solution_items.push_back(curr_item);

        curr_cost -= items_costs[curr_item-1];
    }


    std::reverse(solution_items.begin(), solution_items.end());

    std::string solution = "";
```

```
    for(ll i=0; i<solution_items.size(); i++){

        solution = solution + std::to_string(solution_items[i]) + " ";

    }

    solution.pop_back();


    return solution;
}

void tabulation(std::vector<ll> &items_costs, std::vector<ll> &orders_values, std::vector<ll>
 ↪  &dp_values, std::vector<ll> &dp_reconstruction, ll max_order){

    dp_values[0] = UNIQUE;

    for(ll curr_item = 1; curr_item <= items_costs.size(); curr_item++){
        ll curr_item_value = items_costs[curr_item-1];

        for(ll curr_cost = curr_item_value; curr_cost <= max_order; curr_cost++){

            ll prev_cost = curr_cost-curr_item_value;

            if(dp_values[prev_cost] != IMPOSSIBLE){

                if(dp_values[prev_cost] == AMBIGOUS || dp_values[curr_cost] == AMBIGOUS)
                 ↪  {dp_values[curr_cost] = AMBIGOUS;}
                else if(dp_values[prev_cost] == UNIQUE){

                    if(dp_values[curr_cost] == IMPOSSIBLE){
                        dp_values[curr_cost] = UNIQUE;
                        dp_reconstruction[curr_cost] = curr_item;
                    }
                    else if(dp_values[curr_cost] == UNIQUE){
                        dp_values[curr_cost] = AMBIGOUS;
                    }
                }
            }
        }
    }
}

void restaurantOrders(){

    ll n_items;
    std::cin >> n_items;

    std::vector<ll> items_costs(n_items);
    for(ll i=0; i<n_items; i++){std::cin >> items_costs[i];}

    ll m_orders;
    std::cin >> m_orders;

    ll max_order = LLONG_MIN;
    std::vector<ll> orders_values(m_orders);
```

```
    for(ll curr_order=0; curr_order<m_orders; curr_order++){
        std::cin >> orders_values[curr_order];
        max_order = std::max(max_order, orders_values[curr_order]);
    }

    std::vector<ll> dp_values(max_order+1,IMPOSSIBLE);
    std::vector<ll> dp_reconstruction(max_order+1,-1);

    tabulation(items_costs,orders_values,dp_values,dp_reconstruction,max_order);

    for(ll curr_order = 0; curr_order < m_orders; curr_order++){

        switch(dp_values[orders_values[curr_order]]){
            case IMPOSSIBLE:
                std::cout << "Impossible" << std::endl;
                break;
            case UNIQUE:
                std::cout <<
                ↪ reconstruction(items_costs,dp_reconstruction,orders_values[curr_order]) <<
                ↪ std::endl;
                break;
            case AMBIGOUS:
                std::cout << "Ambiguous" << std::endl;
                break;
        }
    }
}

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

### Presidential Elections

Por el momento saltado (muy largo)

### Spiderman Workout

Staying fit is important for every super hero, and Spiderman is no exception. Every day he undertakes a climbing exercise in which he climbs a certain distance, rests for a minute, then climbs again, rests again, and so on. The exercise is described by a sequence of distances $d_1, d_2, \ldots, d_m$, telling how many meters he is to climb before the first break, before the second break, and so on. From an exercise perspective it does not really matter if he climbs up or down at the $i$:th climbing stage, but it is practical to sometimes climb up and sometimes climb down so that he both starts and finishes at street level. Obviously, he can never be below street level. Also, he would like to use as low a building as possible (he does not like to admit it, but he is actually afraid of heights). The building must be at least 2 meters higher than the highest point his feet reach during the workout.
He wants your help in determining when he should go up and when he should go down. The answer must be legal: It must start and end at street level (0 meters above ground) and it may never go below street level. Among the legal solutions he wants one that minimizes the required building height. When looking for a solution, you may not reorder the distances.
If the distances are 20 20 20 20 20 he can either climb up, up, down, down or up, down, up, down. Both are legal, but the second one is better (in fact optimal) because it only requires a building of height 22, whereas the first one requires a building of height 42. If the distances are 3 2 5 3 1 2, an optimal legal solution is to go up, up, down, up, down, down. Note that for some distance sequences there is no legal solution at all (e.g., for 3 4 2 1 6 4 5).

#### Input

The first line of the input contains an integer $N$ giving the number of test cases, $1 \leq N \leq 101$. The following $2N$ lines specify the test scenarios, two lines per scenario: the first line gives a positive integer $M \leq 40$ which is the number of distances, and the following line contains the $M$ positive integer distances. For any scenario, the total distance climbed (the sum of the distances in that scenario) is at most 1000.

#### Output

For each input scenario a single line should be output. This line should either be the string ÏMPOSSIBLE" if no legal solution exists, or it should be a string of length $M$ containing only the characters Ü" and "D", where the $i$:th character indicates if Spiderman should climb up or down at the $i$:th stage. If there are several different legal and optimal solutions, output one of them (it does not matter which one as long as it is optimal).

```
#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

#define CLIMB_UP true
#define CLIMB_DOWN false

#define MAX_HEIGHT 1000

std::string testcase(std::vector<ll> &distances){

    std::vector<std::vector<ll>> dp_min_jumps(MAX_HEIGHT+1, std::vector<ll>(distances.size()+1,
    ↪  LLONG_MAX)); // dp[altura][estado salto]
    std::vector<std::vector<bool>> dp_directions(MAX_HEIGHT+1,
    ↪  std::vector<bool>(distances.size()+1, CLIMB_DOWN));

    // Caso base:

    dp_min_jumps[0][0] = 0; // En 0 de altura, con ningun salto, te quedas en 0
```

```
    // Iteración

    for(ll curr_distance_idx = 1; curr_distance_idx <= distances.size(); curr_distance_idx++){

        ll curr_distance = distances[curr_distance_idx-1];

        for(ll curr_height = 0; curr_height <= MAX_HEIGHT; curr_height++){

            if(dp_min_jumps[curr_height][curr_distance_idx-1] == LLONG_MAX){continue; } // Si no
            ↪  se pudo llegar desde el paso anterior, no seguir

            // Caso para arriba
            ll new_height = curr_height + curr_distance;
            if(new_height <= MAX_HEIGHT){

                ll posssible_height = std::max(dp_min_jumps[curr_height][curr_distance_idx-1],
                ↪  new_height);

                if(dp_min_jumps[new_height][curr_distance_idx] > posssible_height){
                    dp_min_jumps[new_height][curr_distance_idx] = posssible_height;
                    dp_directions[new_height][curr_distance_idx] = CLIMB_UP;
                }

            }

            // Caso para abajo
            new_height = curr_height - curr_distance;
            if(new_height >= 0){

                ll posssible_height = dp_min_jumps[curr_height][curr_distance_idx-1];

                if(dp_min_jumps[new_height][curr_distance_idx] > posssible_height){
                    dp_min_jumps[new_height][curr_distance_idx] = posssible_height;
                    dp_directions[new_height][curr_distance_idx] = CLIMB_DOWN;
                }

            }

        }

    }

    // Resultado

    if(dp_min_jumps[0][distances.size()] == LLONG_MAX){return "IMPOSSIBLE"; }

    // Reconstrucción

    std::string result;
    ll curr_height = 0;

    for(ll curr_distance_idx = distances.size(); curr_distance_idx >= 1; curr_distance_idx--){
        ll dist = distances[curr_distance_idx-1];

        // Check the direction stored for the optimal path ending at curr_height at this step
        bool move = dp_directions[curr_height][curr_distance_idx];

        if(move == CLIMB_UP){
```

```
            result.push_back('U');
            curr_height -= dist;

        }
        else {
            result.push_back('D');
            curr_height += dist;
        }

    }

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

    return result;

}

int main(){

    ll N_testcases;
    std::cin >> N_testcases;

    for(ll curr_testcase = 0; curr_testcase < N_testcases; curr_testcase++){

        ll M_distances;
        std::cin >> M_distances;

        std::vector<ll> distances(M_distances);
        for(ll curr_distance = 0; curr_distance < M_distances; curr_distance++){ std::cin >>
        ↪  distances[curr_distance]; }

        std::cout << testcase(distances) << std::endl;

    }

    return 0;
}
```

### 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;
}
```

### 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
}
```