# Algoritmos y Estructuras de Datos III

Tomás Agustín Hernández

# Complejidad Computacional (repaso)

## Problema

Descripción de los datos de entrada y la respuesta a proporcionar para cada uno de los datos de entrada.

## Instancia de un Problema

Es un juego válido de datos de entrada.

## Máquina RAM

Supongamos una Máquina RAM.

- La memoria está dada por una sucesión de celdas numeradas. Cada **celda** puede almacenar un valor de **b bits**.
- Supondremos habitualmente que esos **b bits** de cada celda están fijos, y suponemos que todos los datos que maneja el algoritmo se pueden almacenar con **b bits**. **Ej.**: Lo que quiere decir esto es que suponemos que todas las celdas son de 8 bits, y los datos que maneja el algoritmo también son de 8 bits.
- Se tiene un programa imperativo que NO está almacenado en memoria que está compuesto por asignaciones y las estructuras de control habituales.
- Las asignaciones acceden a las celdas de memoria y realizan las operaciones estándar sobre los tipos de datos primitivos habituales.

Cada una de las instrucciones que se ejecuten tienen un tiempo de ejecución asociado

- El acceso a cualquier celda de memoria, tanto lectura como escritura es O(1).
- Las asignaciones y el manejo de las estructuras de control se realiza en O(1).
- Las operaciones entre valores lógicos son O(1).

Las operaciones entre enteros/reales dependen de b

- Las sumas y restas son O(b).
- Las multiplicaciones y divisiones son O(b log b)

**Nota**: Si b está fijo, entonces las operaciones entre enteros/reales es O(1).

## Tiempo de Ejecución de un Algoritmo

Sea A un algoritmo, su tiempo de ejecución es: $T_A(I)$ donde esto indica que es la suma de los tiempos de ejecución del algoritmo en una instancia dada I.
$|I|$: Cantidad de bits necesarios para almacenar los datos de entrada de I.

**Nota**: Si **b está fijo** y la entrada ocupa n celdas de memoria entonces $|I| = bn = O(n)$

## Complejidad de un Algoritmo

Sea A un algoritmo, su complejidad es: $f_A(n) = max_{I:|I|=n}\ T_A(I)$ donde esto indica que la complejidad de un algoritmo A dado un n cualquiera, es el que de mayor tiempo de ejecución tiene en una instancia dada I.

## Cotas

**Cota Superior (O)**: $f(n) \in O(g(n)) \iff \exists c \in R > 0, n_0 \in N \text{ tal que } \forall n \geq n_0 : f(n) \leq c * g(n)$
**Cota Inferior ($\Omega$)**: $f(n) \in \Omega(g(n)) \iff \exists c \in R > 0, n_0 \in N \text{ tal que } \forall n \geq n_0 : f(n) \geq c * g(n)$
**Cota Ajustada ($\theta$)**: $f(n) \in \theta(g(n)) \iff f(n) \in O(g(n)) \text{ y } f(n) \in \Omega(g(n))$. *Es decir,* $\theta(g(n)) = O(g(n)) \cap \Omega(g(n))$

## Tipos de Funciones

- O(n): lineal
- $O(n^2)$: cuadrático
- $O(n^3)$: cúbico
- $O(n^k)\ k \in \mathbb{N}$: polinomial. Ej.: $O(n^4),\ O(n^5)$
- $O(log\ n)$: logarítmico.
- $O(d^n)\ d \in \mathbb{R}_{>1}$: exponencial. Ej.: $O(2^n),\ O(4^n)$

# Algoritmos Satisfactorios y No Satisfactorios

Un Algoritmo Satisfactorio es un algoritmo que tiene un costo menor a otro.
Los algoritmos polinomiales se consideran satisfactorios (cuanto menor sea el grado, mejor).
Los algoritmos supra-polinomiales se consideran no satisfactorios.

# Problema de Optimización

Sea $x \in S$, un problema de optimización consiste en encontrar la mejor solución dentro de un conjunto:

- $z^* = max\ f(x)$
- $z^* = min\ f(x)$

**Función Objetivo**: Es una función de la forma $f : S \implies \mathbb{R}$

- El conjunto S es la **región factible**.
- Los elementos $x \in S$ se llaman **soluciones factibles**.
- El valor $z^x \in \mathbb{R}$ es el **valor óptimo** del problema, y cualquier solución factible $x^* \in S\ /\ f(x^*) = z^x$ se llama un **óptimo** del problema

# Algoritmos de Fuerza Bruta

También llamado búsqueda exhaustiva o generate and test. Genera todas las soluciones factibles y se queda con la mejor (la que cumpla las restricciones que necesitamos). Suele ser fácil de implementar y es un algoritmo exacto: si hay solución, siempre la encuentra.
Lo malo es su complejidad (suele ser exponencial)

**Ej.**: Imaginemos que tenemos un tablero de ajedrez y tenemos que buscar las soluciones en las cuales ninguna dama amenace a otra. Una solución por fuerza bruta sería buscar todas las soluciones que existen, y de ahí agarrar las que me sirvan.

**Ej.**: Imaginemos que tenemos que resolver un Sudoku, si usaramos un casillero de 9x9 y quisieramos aplicar un algoritmo de fuerza bruta, es decir, primero buscar todas las posibles permutaciones 1966270504755529.... posibilidades y de todas estas posibilidades ver cual es solución. Esto es muy tedioso y lento, hay una mejor opción que la fuerza bruta, y es el Backtracking.

# Backtracking

Es una técnica (de fuerza bruta pero mas eficiente) que consiste en una exploración ordenada del espacio de soluciones por medio de la extensión de soluciones parciales **(cuando noto que una solución parcial no me sirve, puedo descartarla e ir a la siguiente sin necesidad de perder más tiempo en esa o sus hijos.)**.

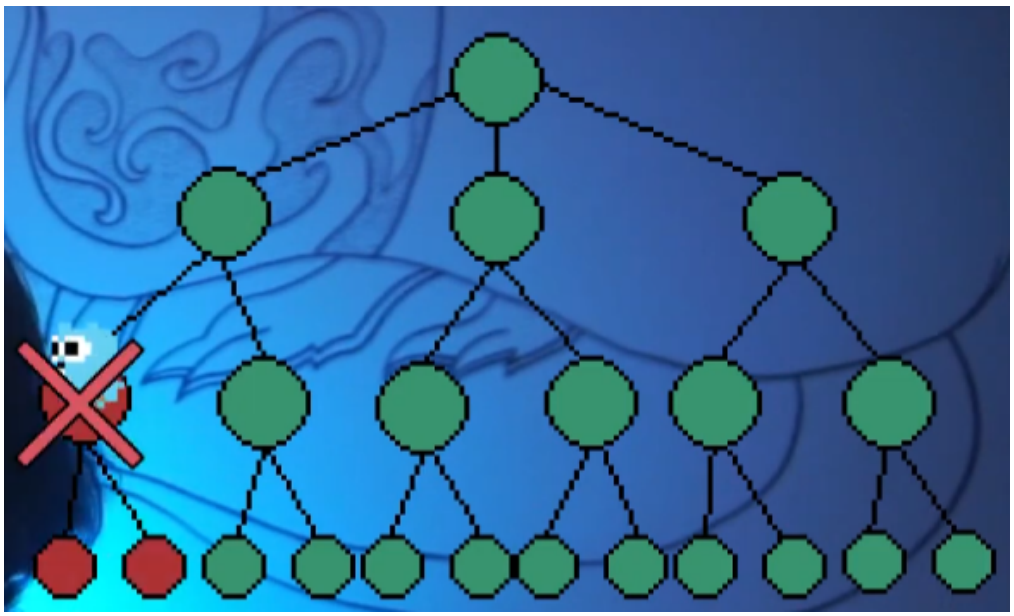
Habitualmente se utiliza un vector $a = (a_1, a_2, ..., a_n)$ para representar una solución candidata, donde cada $a_i \in A_i$ donde $A_i$ es un conjunto ordenado y finito.

El espacio de soluciones final consiste en $A_1\ x\ A_2\ x \ldots\ x\ A_n$.

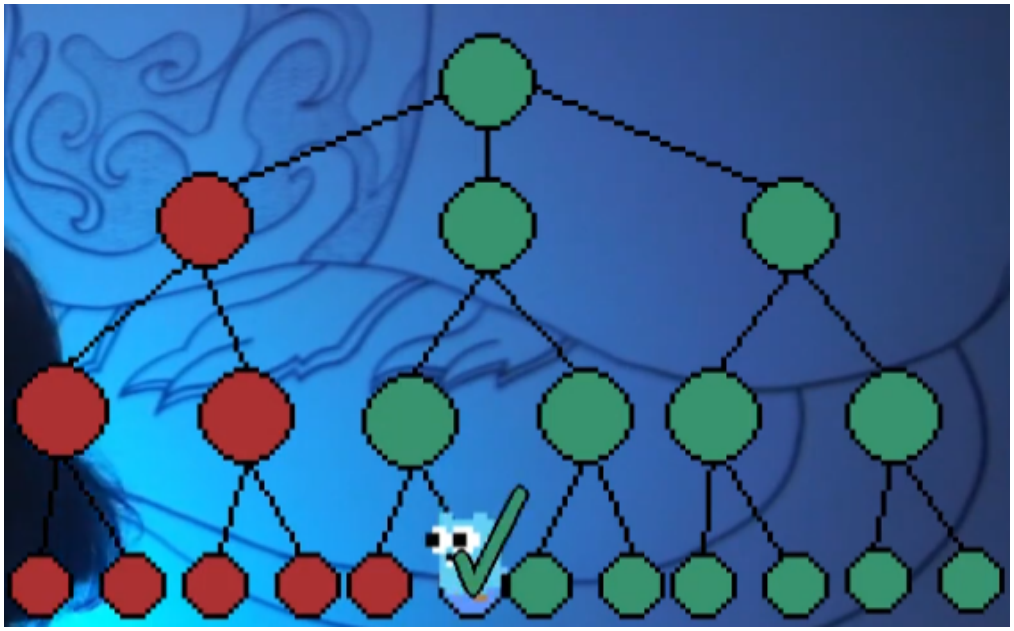
En cada paso se extienden las soluciones parciales $a = (a_1, a_2, ..., a_k)$ con $k < n$ agregando un elemento más que es $a_{k+1} \in S_{k+1} \subseteq A_{k+1}$ al final del vector a. Esto quiere decir que las nuevas soluciones parciales son sucesores directos de la anterior. Si llegamos a un $S_{k+1}$ que es vacío, retrocedemos a la solución parcial $(a_1, a_2, ..., a_{k-1})$.

Esto último de retroceder podemos verlo como un árbol de decisiones, los cuales si un nodo ya de por sì no me sirve, sus hijos tampoco y vuelvo al nodo anterior para seguir evaluando los demás hijos.

**Importante**: Se puede pensar al vector a como un árbol, donde los elementos que hay son vértices. Por lo tanto, cada vértice $y$ tendría vértices hijos $x$ sí y solo sí se puede llegar a $x$ yendo desde $y$.



Comenzamos evaluando desde la raíz, el nodo de la izquierda cumple hasta ahora nuestra posible solución, nos movemos a ese nodo y luego, evaluando su nodo de la izquierda vemos que se rompe, es decir, alguna restricción que pusimos no se cumple, por lo tanto no tiene sentido seguir explorando las demás soluciones.



Vemos que efectivamente, habiendo vuelto a la raíz, ahora sí encontramos un camino mejor que el camino del nodo de la izquierda. Por lo tanto, podemos seguir evaluando hacia abajo.
**Podas por factibilidad**: Evito explorar nodos no factibles.
**Podas por optimalidad**: Evito explorar nodos subóptimos.
**Branch and bound**: Uso la solución más óptima para comparar y ver si exploro eso no.
**Todas las soluciones**:

```
1   algoritmo BT(a,k)
2       si a es solución entonces
3           procesar(a)
4           retornar
5       sino
6           para cada a' en Sucesores(a,k)
7   ----------BT(a', k + 1)
8           fin para
9       fin si
10      retornar
```

**Una solución**:

```
algoritmo BT(a,k)
si a es solución entonces
    sol <- a
    encontro <- true
sino
    para cada a' en Sucesores(a,k)
        BT(a', k + 1)
        si encontro entonces
            retornar
        fin si
    fin para
fin si
retornar
```

**Ej.**: Resolver un sudoku se resuelve en forma muy eficiente con un algoritmo de backtracking.

### Casos de Uso de Backtracking

- Problemas de Decisión.
- Problemas de Optimización: Encontrar la mejor solución.
- Problemas de Enumeración: Todas las soluciones factibles a un problema.

## Programación Dinámica

Consiste en reutilizar valores previamente calculados para ahorrar tiempo. **¿Cuál es el costo?**, el costo es que acá **usamos más espacio de la memoria**.

**Nota**: Si queremos sacar ventaja de la memorización **tenemos que acceder al valor guardado en la estructura de datos en O(1) o en un menor tiempo que lo que costaría calcularlo** otra vez.

Aunque no lo parezca, guardar información que vamos a terminar reutilizando ahorra mucho tiempo.

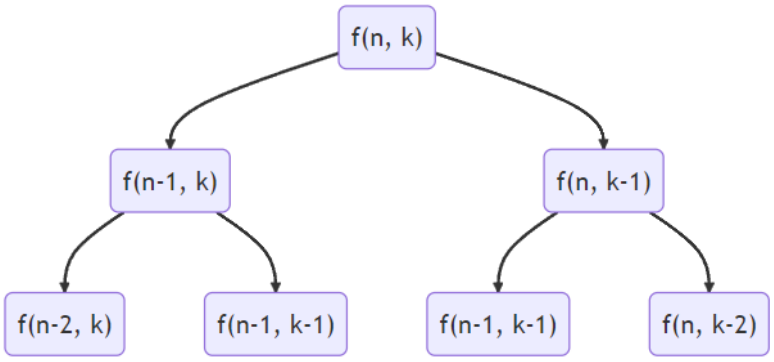
Normalmente, la programación dinámica se utiliza cuando de antemano ya sabemos que vamos a tener que terminar reutilizando valores que ya previamente calculamos para esos parámetros de entrada. A esto se le llama **superposición de estados** y ocurre cuando un árbol de llamadas recursivas resuelve el mismo problema para mismos parámetros de entrada.

Veamos ahora dos formas de encarar un problema de programación dinámica

### Enfoque Top-Down

Se implementa recursivamente. La idea es ir guardando el resultado de cada llamada recursiva en una estructura de datos **(memorización)**. Si llegase a suceder que en una nueva llamada vemos que esa llamada ya ocurrió previamente con los mismos parámetros podemos devolver el valor previamente calculado almacenado en la estructura de datos.

Una forma de visualizarlo es con el árbol de llamados. Si tenemos una función $f(n,k)$ definida como $f(n,k) = f(n-1,k) + f(n,k-1)$ con $k < n$ podríamos representar el pensamiento top-down de la siguiente manera




Claramente podemos observar como $f(n-1,k-1)$ se repite de ambos lados del árbol. Si esto fuese costoso de calcular, estaríamos perdiendo tiempo en algo que podríamos haber guardado.

El nombre de Top-Down viene del lado que partimos del problema más grande y terminamos llegando a un caso base. Véase anexo para ver qué tanto mejora el tiempo de ejecución

### Enfoque Bottom-Up

Generalmente es iterativo, pero no siempre. Empieza resolviendo las partes más pequeñas y fundamentales del problema. Cada vez que resuelve un problema, almacena su solución en una estructura de datos auxiliar.

## Anexo

### Programación Dinámica en C++ (Top-Down)

Con memorización

```
#include <iostream>
#include <vector>
#include <chrono>
using namespace std::chrono;

int fibonacci(int n, std::vector<int> memo){
    if (memo[n] != -1) {
        return memo[n];
    }

    if (n == 0) {
        return 0;
    } else if (n == 1) {
        return 1;
    }

    memo[n] = fibonacci(n - 1, memo) + fibonacci(n - 2, memo);
    return memo[n];
}

int main() {
    auto start = high_resolution_clock::now();
    int n = 0;
    std::cout << "Ingrese un numero para calcular fibonacci" << std::endl;
    std::cin >> n;
    std::vector<int> memo(n+1, -1);
    int fibo = fibonacci(n, memo);
    std::cout << "Fibonacci de " << n << " es " << fibo << std::endl;

    //chrono
    auto stop = high_resolution_clock::now();
    auto duration = duration_cast<microseconds>(stop - start);
    std::cout << "El tiempo de ejecucion es: " << duration.count() << " milisegundos" << std::endl;

    return 0;
}
```

Fibo 40: ? milisegundos.

Sin memorización

```
#include <iostream>
#include <vector>
#include <chrono>
using namespace std::chrono;


int fibonacci(int n){

    if (n == 0) {
        return 0;
    } else if (n == 1) {
```

```
12              return 1;
13          }
14
15          return fibonacci(n - 1) + fibonacci(n - 2);
16      }
17
18      int main() {
19          auto start = high_resolution_clock::now();
20          int n = 0;
21          std::cout << "Ingrese un numero para calcular fibonacci" << std::endl;
22          std::cin >> n;
23          int fibo = fibonacci(n);
24          std::cout << "Fibonacci de " << n << " es " << fibo << std::endl;
25
26          //chrono
27          auto stop = high_resolution_clock::now();
28          auto duration = duration_cast<microseconds>(stop - start);
29          std::cout << "El tiempo de ejecucion es: " << duration.count() << " milisegundos" << std::endl;
30
31          return 0;
32      }
```

Fibo 40: ? milisegundos.