# 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 posibles combinaciones sin importar si se arma una solución correcta o no. Una vez que tiene todas las posibles soluciones, elige de aquellas cuales son las que me sirven para resolver mi problema.
- Lo malo de la fuerza bruta es que de antemano, para saber qué soluciones son correctas debemos encontrar **todas** inclusive las que no son correctas y esto es súper costoso.
- Fácil de implementar.
- Algoritmo exacto. Si $\exists$ solución, la encuentra.
- Es útil para algoritmos que sabemos que la entrada es pequeña y para verificar correctitud de las soluciones.
- Complejidad: 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 existe a cada partida de ajedrez, 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.

## Esquema de Fuerza Bruta

```
1 para cada x en S hacer:
2     si P(x) entonces
3         procesar x
```

Básicamente la idea es primero buscar S, S es el conjunto de posibilidades (combinaciones) que hay.
Cada x sería un elemento de S (x es un subconjunto), es decir, una combinación que es probable solución.
P(x) verifica si la condición que buscamos se cumple dado el subconjunto de S.
Procesar x agrega el subconjunto como solución.

# Backtracking

Es una técnica (de fuerza bruta pero mas eficiente) que consiste en armar un árbol donde debemos extender soluciones parciales **(cuando noto que una solución parcial (un camino) 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.
En cada paso, se extiende este **vector a** y se va formando una nueva **solución parcial** $a = (a_1, a_2, ..., a_k)$ 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** $(a_1, a_2, ..., a_{k-1})$

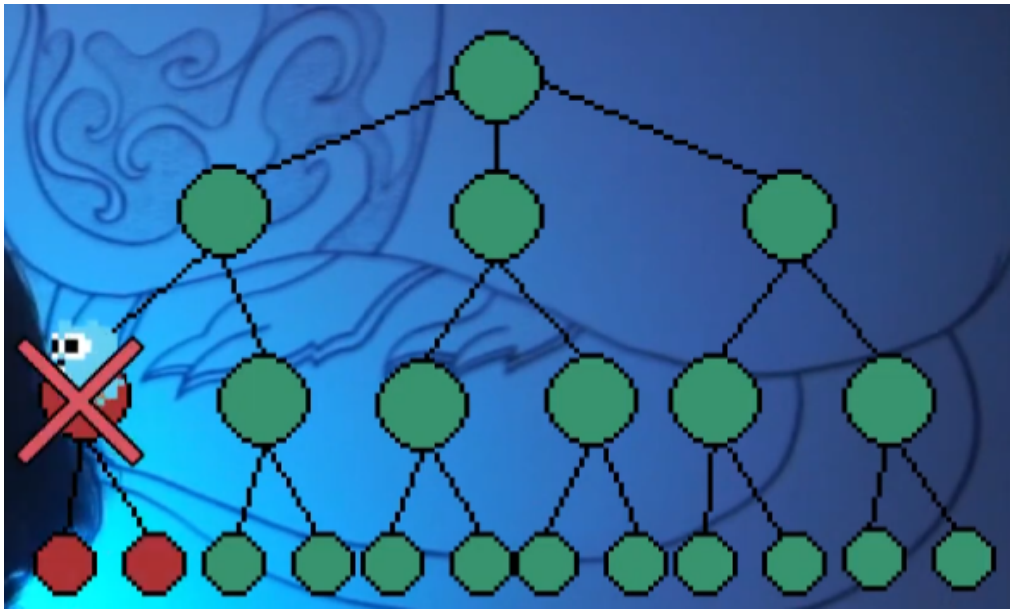
El espacio de soluciones final consiste en $A_1 \ x \ A_2 \ x \dots \ x \ A_n$.

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.
Entonces...

- Conjunto de Soluciones Candidatas: Son todas las combinaciones posibles de los elementos. Pueden ser válidas o no. Son aquellas que habitan el último nivel del árbol (no se pueden extender más) pues no quedan más decisiones por tomar más que considerarlas válidas o inválidas en nuestro problema. Son las hojas del árbol.
- Conjunto de Soluciones Parciales: Son todas las combinaciones posibles de los elementos hasta el anteúltimo nivel (si fuese del último sería una solución candidata). Acá se consideran todos los subconjuntos que se van armando en cada paso. Si llegase a haber uno que excede o no cumple lo que necesitamos, no es una solución parcial.
- Conjunto de Soluciones Válidas: Son aquellas Soluciones Candidatas que cumplen con lo que buscamos.

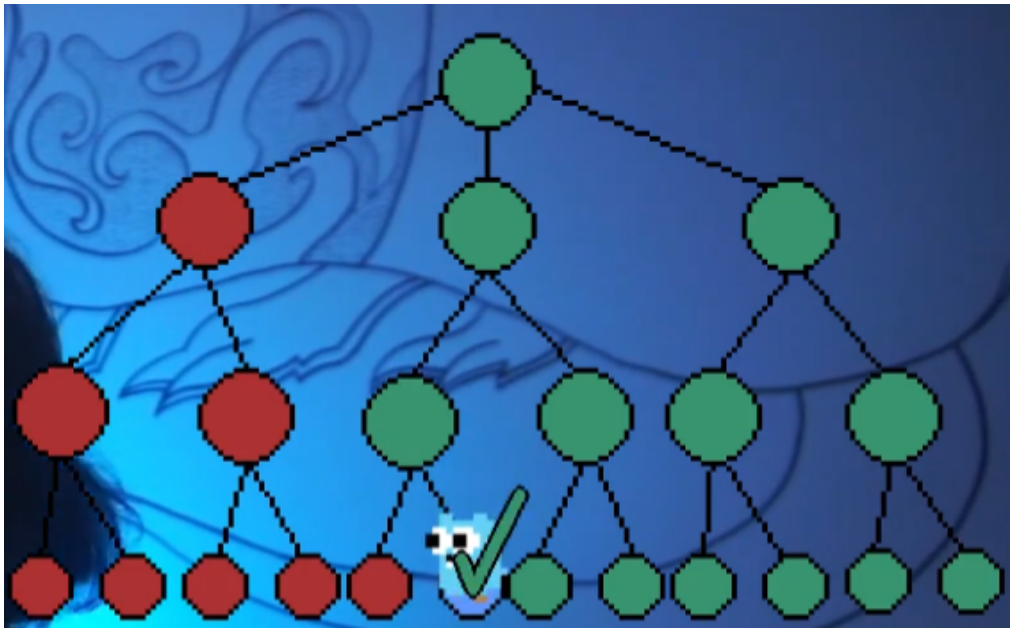
Preguntar:

- ¿El conjunto de soluciones parciales incluye a quien no es una solucion válida?
- ¿El conjunto de soluciones parciales incluye a las candidatas? En el ej. 1 parece que sí por como lo definen.



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.
Véase anexo para un ejemplo didáctico.

## Podas

Nos permiten descartar configuraciones.
**Podas por factibilidad**: Evito explorar nodos no factibles. Esto quiere decir que:

- Si veo que al evaluar el próximo nodo del camino ya no es solución retrocedo inmediatamente al paso anterior.
- Esto de volver al paso anterior, básicamente lo que estoy haciendo es matar al nodo que no me sirve y a los hijos de ese nodo. Si tengo $\{a_1 \to a_2 \to a_3\}$ pero a la hora de evaluar veo que $a_1$ no me sirve, por dominó tampoco me sirven $a_2$ ni $a_3$

**Podas por optimalidad**: Evito explorar nodos subóptimos. Esto quiere decir que:

- Si veo que al evaluar el próximo nodo del camino, ya no forma la mejor solución entonces retrocedo inmediatamente al paso anterior.
- **Importante**: En las podas por optimalidad se necesita un criterio más para saber qué seria considerada una mejor solución que otra. Un ejemplo puede ser un problema de un viajante de comercio que debe encontrar el camino de menor costo para visitar todas las ciudades. A medida que encuentro una solución entera, puedo ir buscando otras y comparándolas. Si en algún momento veo que ya se pasó (tiene un costo mayor que la solución anterior encontrada) de la solución entera la descarto. Es algo más estimativo.

El éxito del backtracking depende de las podas.
**Branch and bound**

- Separo en subproblemas con un límite inferior/superior del valor óptimo que puede obtenerse en ese subproblema.
- La poda se realiza al comparar los límites inferior/superior de los subproblemas con el mejor valor óptimo conocido. Si el límite inferior/superior de un subproblema es peor que la mejor solución conocida, ese subproblema es podado.
- El objetivo de branch and bound es encontrar la solución más óptima al problema.

Backtracking garantiza encontrar al menos una solución válida y luego optimiza, mientras que branch and bound está diseñado para encontrar la mejor solución posible de manera más eficiente desde el principio utilizando límites para podar ramas del espacio de búsqueda.

## Soluciones

**Todas las soluciones**:

```
algoritmo BT(a,k)
    si a es solución entonces
        procesar(a)
        retornar
    sino
        para cada a'-en-Sucesores(a,k)
            BT(a', k + 1)
        fin para
    fin si
    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. **Importante**: Recordar que casi siempre, para ir guardando las soluciones tenemos que crear un vector de igual longitud que la data que tenemos que evaluar, y ese vector de solucion parcial que se manda por parametros tenemos que ir colocando el caso que sea 1 (esta) y el caso 0 (no está).

### 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

La idea de la Programación Dinámica es dividir un problema en subproblemas de tamaños menores que son más fáciles de resolver. Una vez resueltos estos subproblemas, se combinan las soluciones obtenidas para generar la solución del problema original.

- Es aplicado típicamente a problemas de optimización combinatoria.
- Resulta adecuado para algunos problemas de naturaleza recursiva.
- Consiste en reutilizar valores (resueltos por otros subproblemas) previamente calculados para ahorrar tiempo (es decir, evitamos repetir llamadas recursivas que tengan los mismos parámetros). **¿Cuál es el costo?**, el costo es que acá **usamos más espacio de la memoria**.
- Esquema de **memoización**

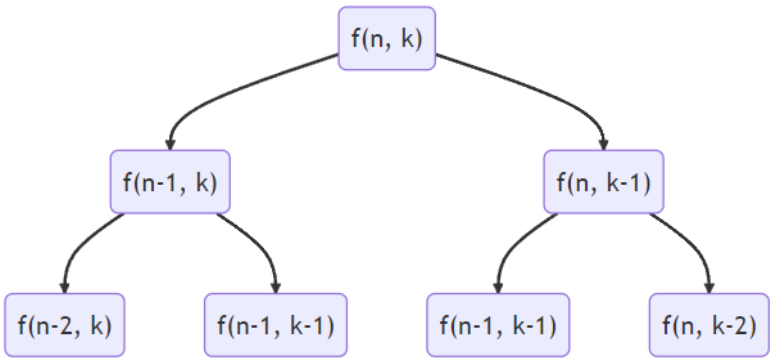
**Nota**: Si queremos sacar ventaja de la memorización **tenemos que acceder al valor guardado en la estructura de datos en O(1) o en un menor tiempo que lo que costaría calcularlo** otra vez. Aunque no lo parezca, guardar información que vamos a terminar reutilizando ahorra mucho tiempo.

**Importante**: No es necesario guardar todos los resultados de todos los subproblemas. Muchas veces podemos identificar cuales no serían necesarios reutilizar.

### 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

Veamos un ejercicio para calcular el número combinatorio, y resolvamoslo Top-Down y Bottom-up.

- **Definición:** Si $n \geq 0$ y $0 \leq k \leq n$, se define

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

- **Teorema:** Si $n \geq 0$ y $0 \leq k \leq n$, entonces

$$\binom{n}{k} = \begin{cases} 1 & \text{si } k = 0 \text{ o } k = n \\ \binom{n-1}{k-1} + \binom{n-1}{k} & \text{si } 0 < k < n \\ 0 & \text{caso contrario} \end{cases}$$

```
combiRec(n, k, T)
entrada: n, k ∈ N
salida: (n k)

si T[n][k] = NULL hacer
  si k = 0 o k = n hacer
    T[n][k] ← 0
  else
    T[n][k] ← combiRec(n − 1, k − 1, T) + combiRec(n − 1, k, T)
  fin si
fin si
retornar T[n][k]
```

En este enfoque, lo malo que tenemos es que vamos memorizando todos los resultados que no tengo guardados y toma mucho más espacio, y ni siquiera sabemos si en algún momento lo vamos a reutilizar.

### 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.
La imagen de abajo representa una solución al problema del número combinatorio resuelto por Bottom-Up. Es mucho más eficiente que Top-Down porque vemos que solo nos basta conocer la fila anterior (los últimos 2 valores) para determinar la siguiente fila.
Esto es mucho más óptimo que hacerlo de manera Top-Down.

| $n \backslash k$ | 0 | 1 | 2 | 3 | 4 | $\cdots$ | $k-1$ | $k$ |
|---|---|---|---|---|---|---|---|---|
| 0 | 1 | | | | | | | |
| 1 | 1 | 1 | | | | | | |
| 2 | 1 | 2 | 1 | | | | | |
| 3 | 1 | 3 | 3 | 1 | | | | |
| 4 | 1 | 4 | 6 | 4 | 1 | | | |
| $\vdots$ | $\vdots$ | | | | | $\ddots$ | | |
| $k-1$ | 1 | | | | | | 1 | |
| $k$ | 1 | | | | | | | 1 |
| $\vdots$ | $\vdots$ | | | | | | | |
| $n-1$ | 1 | | | | | | | |
| $n$ | 1 | | | | | | | |

**algoritmo** *combinatorio*$(n,k)$
**entrada:** dos enteros $n$ y $k$
**salida:** $\binom{n}{k}$

**para** $i = 1$ **hasta** $n$ **hacer**
  $A[i][0] \leftarrow 1$
**fin para**
**para** $j = 0$ **hasta** $k$ **hacer**
  $A[j][j] \leftarrow 1$
**fin para**
**para** $i = 2$ **hasta** $n$ **hacer**
  **para** $j = 2$ **hasta** $\min(i-1, k)$ **hacer**
    $A[i][j] \leftarrow A[i-1][j-1] + A[i-1][j]$
  **fin para**
**fin para**
**retornar** $A[n][k]$

## Divide & Conquer

Esta técnica consiste en dividir **un problema en subproblemas del mismo tipo** que en el original, resolver los subproblemas y luego combinar las soluciones.

**Importante**: Cuando hablamos de subproblemas, hablamos de subproblemas necesariamente más chicos que el problema original; Deben ser todos del mismo tipo y no podemos resolver diferentes subproblemas.

Ej.: Si tengo una pared enorme, la puedo pintar por partes. Si a la primera la pinté de arriba hacia abajo la segunda la voy a pintar igual y exactamente con los mismos movimientos. Al final de todo, veré que la pared quedó perfectamente pintada. El ejemplo representa claramente un problema de Divide & Conquer pues

- Al ser un problema grande, divido la tarea de pintar la pared enorme en partes iguales de una manera más sencilla y llevadera.
- Las paredes las pinto a todas de arriba hacia abajo y con exactamente los mismos movimientos (subproblemas del mismo tipo).
- Como sé que la porción de pared anterior quedó perfectamente pintada y la siguiente también lo estará, entonces terminaré pintando la pared entera.

**Importante**: Divide & Conquer debería tener una complejidad parecida a resolver el problema original. Si resolver el problema original nos cuesta $O(n)$ y con esta técnica $O(n^3)$ algo anda mal porque el tiempo de resolución debería ser igual (o menor) separando y resolviendo los subproblemas.

### Teorema Maestro

Nos permite calcular la complejidad temporal de un un algoritmo que involucra recursión.

$$T(n) = \begin{cases} a * T(n/c) + f(n) & si \quad n > 1 \\ 1 & si \quad n = 1 \end{cases}$$

Donde:

- a: cantidad de subproblemas en los que el problema se divide.
- n/c: es el tamaño de cada subproblema.
- c: cantidad de problemas.
- T(n/c): Es el tiempo de ejecución del algoritmo para un problema de tamaño n/c.
- f(n): Costo de dividir el problema y combinar los resultados de los subproblemas.

### Casos del Teorema Maestro

- Si $f(n) = O(n^{log_c a - \epsilon})$ con $\epsilon > 0$, entonces $T(n) = \Theta(n^{log_c\ a})$
- Si $f(n) = \Theta(n^{log_c a})$ entonces $T(n) = \Theta(n^{log_c a}\ log\ n)$
- Si $f(n) = \Omega(n^{log_c\ a+\epsilon})$ para $\epsilon > 0$, y si $af(\frac{n}{c}) < kf(n)$ para $k < 1$ y para n suficientemente grandes, entonces $T(n) = \Theta(f(n))$

### Qué no puede pasar en el Teorema Maestro

- f(n) no puede ser negativo porque es el tiempo que tarda el algoritmo. Siempre es positivo.
- a es una constante positiva pues el número de subproblemas una vez que arranca la recursión es fija.

## Heurísticas

Es un procedimiento computacional que intenta obtener soluciones de buena calidad para un problema y que su comportamiento sea preciso.
Decimos que A es un algoritmo $\epsilon$-aproximado ($\epsilon \geq 0$) para un problema si

$$|\frac{X_A - X_{OPT}}{X_{OPT}}| \leq \epsilon$$

# Algoritmos Golosos (Algoritmos Greedy)

Es una técnica que consiste en elegir en cada paso la mejor opción sin considerar lo que pueda pasar después.
Saber cual es la mejor opción depende de qué nos pidan como prioritario.
**¿A qué me refiero como prioritario?**

- Si te pido que de una lista de 100 películas, tomes la mayor cantidad de películas para que yo vea en 8hs: La idea sería que tomes las películas con menor duración (porque a menor duración, mas películas vas a elegir)
- Si te pido que vayas a la panadería y traigas 4kg de pan: No se puede resolver con greedy, porque no te especifico cuál sería el pan ideal.

Veamos algunos ejemplos más

- Tengo monedas de 1, 5, 10 y 25 centavos. Debo dar 0,69 de vuelto con la **menor cantidad de monedas**.
  - IDEA: La mejor elección en este caso que tengo que devolver menor cantidad de monedas, me conviene devolver más monedas de la moneda con mayor valor.
  - Toma 1 moneda de 25 pues sumando lo que tengo sucede que $0{,}25 < 0{,}69$.
  - Toma 1 moneda de 25 pues sumando lo que tengo sucede que $0{,}50 < 0{,}69$
  - Toma 1 moneda de 25 $0{,}75 < 0{,}69$ es falso, por lo que sigo teniendo 0,50.
  - Toma 1 moneda de 10 pues sumando lo que tengo sucede que $0{,}60 < 0{,}69$.
  - Toma 1 moneda de 10 $0{,}70 < 0{,}69$ es falso, por lo que sigo teniendo 0,60.
  - Toma 1 moneda de 5 pues sumando lo que tengo sucede que $0{,}65 < 0{,}69$.
  - Toma 1 moneda de 5 $0{,}70 < 0{,}69$ es falso, por lo que sigo teniendo 0,65.
  - Toma 1 moneda de 1 (4 veces) pues sumando lo que tengo sucede que $0{,}69 < 0{,}69$
- Un servidor tiene n clientes para atender, los puede atender en cualquier orden. El objetivo es determinar en que orden se deben atender los clientes para **minimizar la suma de los tiempos de espera** de los clientes.
  - Me pongo en situación: Supongamos que tenemos 1000 clientes. Tenemos 2 representantes, 2 de los clientes tienen problemas que se resuelven en 4hs. Los demás, tienen problemas pero se resuelven en poco tiempo. IDEA: Ordeno los clientes por tiempo de demora de sus trámites y voy resolviendo los más cortos. Esto producirá que los clientes se vayan contentos porque se los atiende rápido (pues sabemos que a la larga, la suma de los tiempos de los trámites de los clientes con trámites cortos, es igual o menor a atender a solo los 2 clientes).
  - Mal camino: Si priorizaramos a los clientes que tienen trámites más largos tendríamos 998 clientes enojados.
  - Buen camino: Si priorizaramos a los 998 clientes aunque los otros dos hayan llegado antes, seguramente ellos estén enojados pero es mejor tener 998 felices que solo 2.

Esto demuestra que los Algoritmos Golosos nos dan solución para **minimizar el tiempo total de espera en un sistema atendiendo por menor tiempo de atención** y también **minimizar la cantidad de monedas que devolvemos, pero seguramente tendremos más monedas de mayor valor**.

# Grafos

# Anexo

## Backtracking

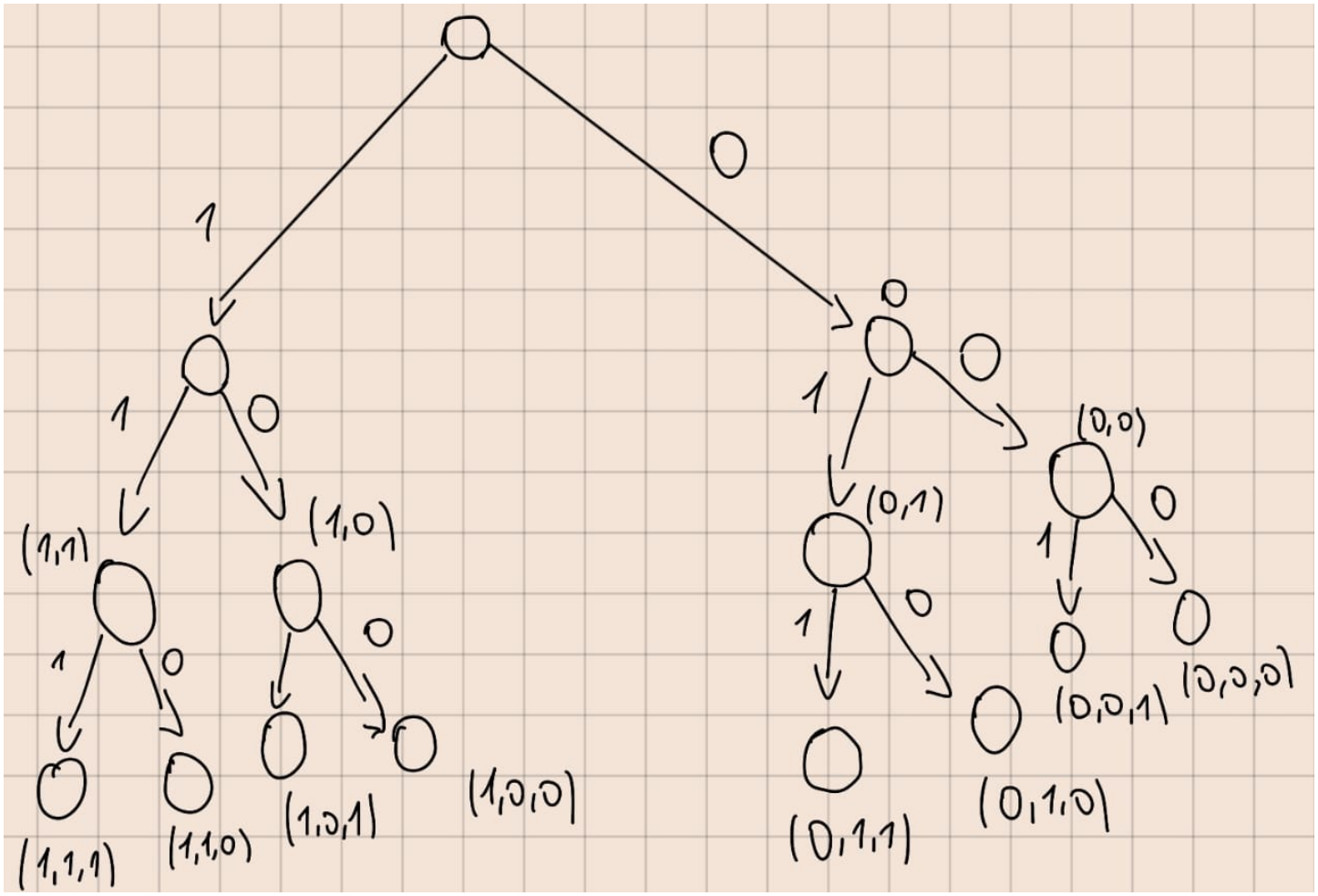
1. En este ejercicio vamos a resolver el problema de suma de subconjuntos con la técnica de *backtracking*. Dado un multiconjunto $C = \{c_1, \ldots, c_n\}$ de números naturales y un natural $k$, queremos determinar si existe un subconjunto de $C$ cuya sumatoria sea $k$. Vamos a suponer fuertemente que $C$ está ordenado de alguna forma arbitraria pero conocida (i.e., $C$ está implementado como la secuencia $c_1, \ldots, c_n$ o, análogamente, tenemos un iterador de $C$). Las *soluciones (candidatas)* son los vectores $a = (a_1, \ldots, a_n)$ de valores binarios; el subconjunto de $C$ representado por $a$ contiene a $c_i$ si y sólo si $a_i = 1$. Luego, $a$ es una solución *válida* cuando $\sum_{i=1}^{n} a_i c_i = k$. Asimismo, una *solución parcial* es un vector $p = (a_1, \ldots, a_i)$ de números binarios con $0 \leq i \leq n$. Si $i < n$, las soluciones *sucesoras* de $p$ son $p \oplus 0$ y $p \oplus 1$, donde $\oplus$ indica la concatenación.

   *a*) Escribir el conjunto de soluciones candidatas para $C = \{6, 12, 6\}$ y $k = 12$.

   *b*) Escribir el conjunto de soluciones válidas para $C = \{6, 12, 6\}$ y $k = 12$.

   *c*) Escribir el conjunto de soluciones parciales para $C = \{6, 12, 6\}$ y $k = 12$.

El ejercicio nos plantea la idea de que necesitamos de alguna manera saber qué combinaciones de subconjuntos son válidas tal que la suma de sus elementos internos da k = 12.
Por lo que se nos indica, cada una de estas combinaciones está formada de manera binaria, es decir, siendo $C = \{6, 12, 6\}$ si yo tengo el subconjunto $\{1, 0, 0\}$ eso indicaría que la suma que da los elementos del subconjunto es 6.
Lo primero que tenemos que hacer es plantear el árbol entero ¿por qué? porque necesitamos de alguna manera ver **todas** las soluciones candidatas que tenemos.



Así, las **Soluciones Candidatas** son las del último nivel:
Por lo tanto $\{(1,1,1),(1,1,0),(1,0,1),(1,0,0),(0,1,1),(0,1,0),(0,0,1),(0,0,0)\}$
Luego, las **Soluciones Parciales** son aquellas que hasta el anteúltimo nivel son válidas:
Por lo tanto $\{(0),(1),(1,0),(0,1),(0,0),(1,1)\}$
Por último, las **Soluciones Válidas** son aquellas que cumplen con nuestra condición en las Soluciones Candidatas.
Por lo tanto $\{(0,1,0),(1,0,1)\}$

En este ejercicio hacemos podas por factibilidad pues, si la suma de los elementos excede 12 todo el camino que sigue es inválido.

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

**Importante**: Notar que estoy usando long long porque los números que puede tomar fibonacci son tan grandes, que evito correr riesgos con int.
**Importante**: Notar que estoy envíando el vector por referencia. De lo contrario, se generaría uno nuevo en cada llamada.

Con memorización

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

long long fibonacci(long long n, std::vector<long long>& 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<long long> memo(n + 1, -1);
    long long 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 50 (12586269025): 2887439 milisegundos.

Sin memorización

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

long long fibonacci(int n)
{

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

    return fibonacci(n - 1) + fibonacci(n - 2);
}

int main()
{
    auto start = high_resolution_clock::now();
    int n = 0;
    std::cout << "Ingrese un numero para calcular fibonacci" << std::endl;
    std::cin >> n;
    long long fibo = fibonacci(n);
    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 50: ? milisegundos. No termina, se cuelga.