OVERVIEW DE ALGORITMOS

1.1. Algoritmos de Fuerza Bruta.

Los algoritmos de Fuerza bruta buscan hallar la solución a un problema generando cada uno de los posibles candidatos para la misma, y verificando si efectivamente cada uno de éstos cumple las restricciones o condiciones para ser la solución buscada. En caso de que exista alguna solución al problema, un algoritmo de fuerza bruta siempre la hallará. Estos algoritmos son sencillos de implementar, pero muy ineficientes en lo que refiere a tiempo de computo o ejecución (algoritmos de orden exponencial en general), que perfectamente pueden dejar "colgado" el proceso en ejecución, por más veloces que sean los procesadores que se tenga.

Ejemplo sencillo: el problema de descifrar un password. Para simplificar, supongamos que se tiene un password numérico secreto de 4 cifras que queremos descubrir.

Para cada cifra tenemos 10 dígitos posibles (0…9),
lo que daría una cantidad de $10 * 10 * 10 * 10 = 10^4 = 10000$ posibilidades,

Si bien este chequeo para una máquina actual lleva un tiempo ínfimo, veamos qué pasa si ampliamos el rango de caracteres admisibles y la longitud para formarlo.

Si se consideran 8 caracteres para la longitud del password, y se admite además de los dígitos numéricos, 28 letras del alfabeto, la cantidad de posibilidades aumentaría a:

$$38 * 38 * 38 * 38 * 38 * 38 * 38 * 38 = 38^8 = 4.347.792.138.496 \text{ posibilidades}$$

Como vemos, la cantidad de candidatos a soluciones aumenta de forma exponencial al agregar más posibilidades para los caracteres admisibles en el password (y eso que no estamos considerando diferencia entre minúsculas y mayúsculas, ni caracteres especiales).

Si bien la búsqueda por fuerza bruta se sigue utilizando, solo se recomienda cuando se tiene una cantidad no muy elevada de candidatos, o cuando se pueden reducir las posibilidades iniciales para los mismos.

1.2. Algoritmos Voraces.

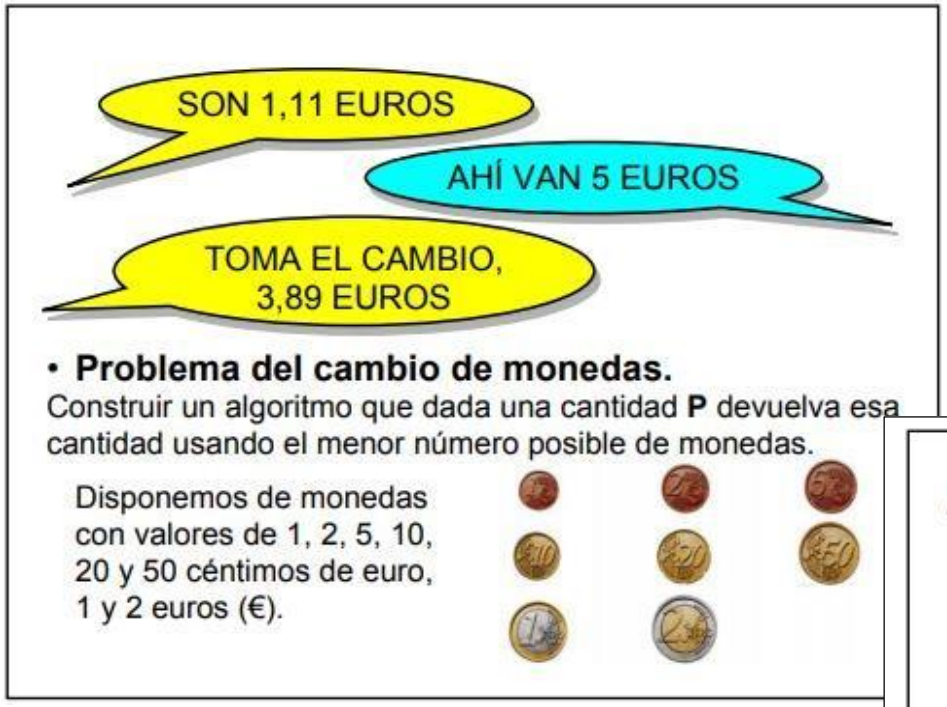
Los algoritmos voraces ó ávidos (en inglés “greedy”) funcionan en fases. En cada fase, se toma una decisión que parece buena, sin considerar las consecuencias futuras. En general, esto significa que se escoge algún optimo local. Esta estrategia de “tomar lo que se pueda ahora” es de donde proviene el nombre de esta clase de algoritmos.

Cuando el algoritmo termina, esperamos que optimo local sea igual a optimo global. Si este es el caso, el algoritmo es correcto, si no, el algoritmo ha producido una solución subóptima. Si no se requiere la mejor respuesta absoluta, se usan a veces algoritmos ávidos simples para generar respuestas aproximadas, en vez de emplear los algoritmos complejos que suelen requerirse para generar una respuesta exacta.

Es decir, en un algoritmo ávido o voraz, en cada punto de decisión se selecciona la opción que tiene el coste inmediato más pequeño (local), sin intentar considerar si esta opción es parte de una solución óptima para el problema completo (una solución global).

Ejemplo: el cambio de monedas.

Si ud. está (por estudio o vacaciones) en un país de la Unión Europea…



- **Problema del cambio de monedas.**

Construir un algoritmo que dada una cantidad **P** devuelva esa cantidad usando el menor número posible de monedas.

Disponemos de monedas con valores de 1, 2, 5, 10, 20 y 50 céntimos de euro, 1 y 2 euros (€).

- **Caso** Devolver 3,89 Euros.

1 moneda de 2€, 1 moneda de 1€, 1 moneda de 50 c€, 1 moneda de 20 c€, 1 moneda de 10 c€ , 1 moneda de 5 c€ y 2 monedas de 2 c€. Total: 8 monedas.

- El método intuitivo se puede entender como un **algoritmo voraz**: en cada paso añadir una moneda nueva a la solución actual, hasta llegar a **P**.

Método general.

Problema del cambio de monedas

- **Conjunto de candidatos**: todos los tipos de monedas disponibles. Supondremos una cantidad ilimitada de cada tipo.
- **Solución**: conjunto de monedas que sumen **P**.
- **Función objetivo:** minimizar el número de monedas.

Representación de la solución:

- **$(x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8)$**, donde **$x_i$** es el número de monedas usadas de tipo **i**.
- Suponemos que la moneda **i** vale **c_i**.
- **Formulación:**

$$\text{Minimizar} \sum_{i=1\ldots8} x_i, \quad \text{sujeto a} \sum_{i=1\ldots8} x_i \cdot c_i = P, \quad x_i \geq 0$$

Método general.

Esquema general de un algoritmo voraz:

```
voraz (C: CjtoCandidatos; var S: CjtoSolución)
    S:= Ø
    mientras (C ≠ Ø) Y NO solución(S) hacer
        x:= seleccionar(C)
        C:= C - {x}
        si factible(S, x) entonces
            insertar(S, x)
        finsi
    finmientras
    si NO solución(S) entonces
        devolver "No se puede encontrar solución"
    finsi
```

Método general.

Funciones genéricas

- **solución (S).** Comprueba si un conjunto de candidatos es una solución (independientemente de que sea óptima o no).
- **seleccionar (C)**. Devuelve el elemento más "prometedor" del conjunto de candidatos pendientes (no seleccionados ni rechazados).
- **factible (S, x)**. Indica si a partir del conjunto **S** y añadiendo **x**, es posible construir una solución (posiblemente añadiendo otros elementos).
- **insertar (S, x)**. Añade el elemento **x** al conjunto solución.

Método general.

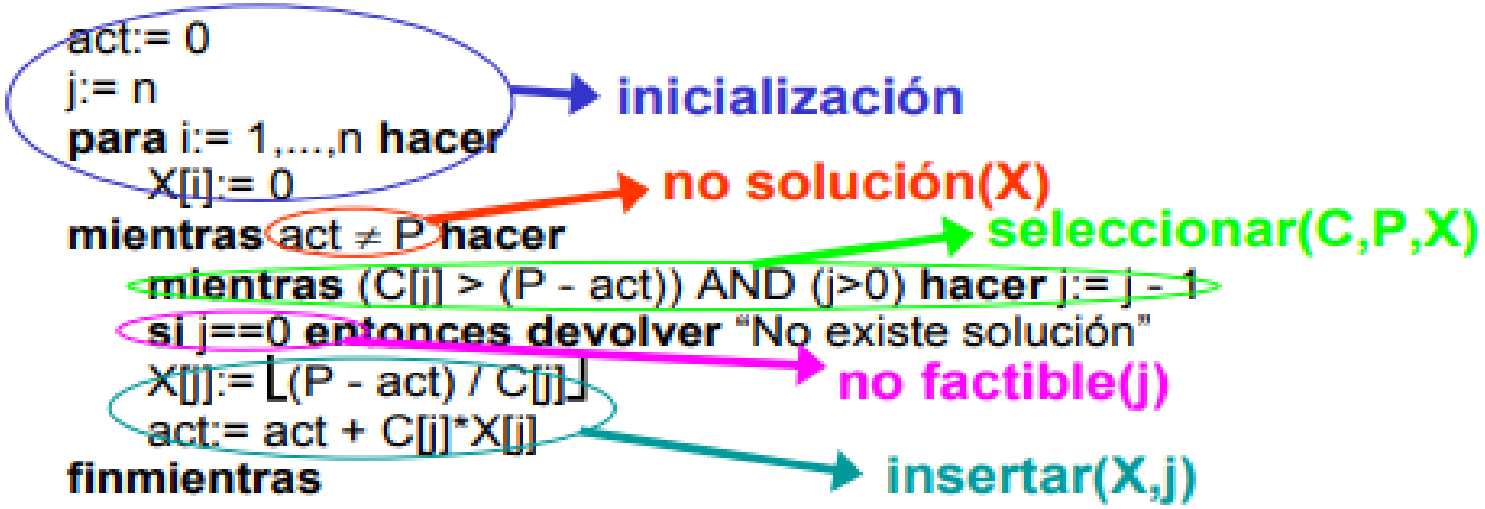
Funciones del esquema:

- **inicialización.** Inicialmente $x_i = 0$, para todo i= 1..8
- **solución**. El valor actual es solución si $\sum x_i \cdot c_i = P$
- **seleccionar**. ¿Qué moneda se elige en cada paso de entre los candidatos?
- **Respuesta:** elegir en cada paso la moneda de valor más alto posible, pero sin sobrepasar la cantidad que queda por devolver.
- **factible**. Valdrá siempre verdad.
- En lugar de seleccionar monedas de una en una, usamos la división entera y cogemos todas las monedas posibles de mayor valor.

Método general.

- **Implementación.** Usamos una variable local **act** para acumular la cantidad devuelta hasta este punto.
- Suponemos que las monedas están ordenadas de menor a mayor valor.

```
DevolverCambio (P: entero; C: array [1..n] de entero;
    var X: array [1..n] de entero)
    act:= 0
    j:= n
    para i:= 1,...,n hacer
        X[i]:= 0
    mientras act ≠ P hacer
        mientras (C[j] > (P - act)) AND (j>0) hacer j:= j - 1
        si j==0 entonces devolver "No existe solución"
        X[j]:= ⌊(P - act) / C[j]⌋
        act:= act + C[j]*X[j]
    finmientras
```



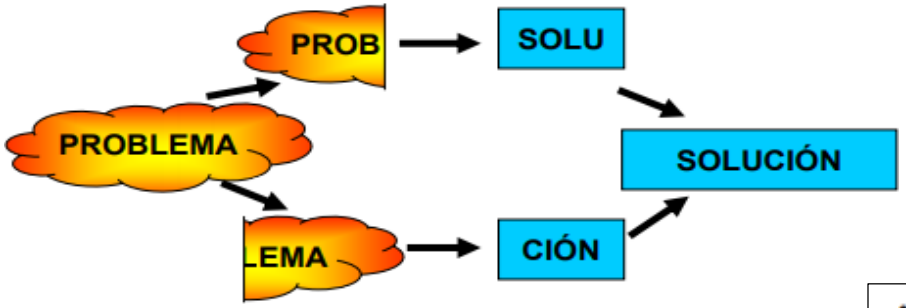
Análisis de tiempos de ejecución.

- El orden de complejidad depende de:
  - El número de candidatos existentes.
  - Los tiempos de las funciones básicas utilizadas.
  - El número de elementos de la solución.
  - ...
- **Ejemplo. n**: número de elementos de C. **m**: número de elementos de una solución.
- Repetir, como máximo **n** veces y como mínimo **m**:
  - Función solución: **f(m)**. Normalmente O(1) ó O(m).
  - Función de selección: **g(n)**. Entre O(1) y O(n).
  - Función **factible** (parecida a **solución**, pero con una solución parcial): **h(m)**.
  - Inserción de un elemento: **j(n, m)**.

1.3. Algoritmos “Divide y Vencerás”.

Método general

- La técnica **divide y vencerás** consiste en:
  - Descomponer un problema en un conjunto de subproblemas más pequeños.
  - Se resuelven estos subproblemas.
  - Se combinan las soluciones para obtener la solución para el problema original.



- **Esquema general:**

DivideVencerás (p: problema)

Dividir (p, p_1, p_2, ..., p_k)
para i:= 1, 2, ..., k
s_i:= *Resolver* (p_i)
solución:= *Combinar* (s_1, s_2, ..., s_k)

- Normalmente para resolver los subproblemas se utilizan llamadas recursivas al mismo algoritmo (aunque no necesariamente).

- **Ejemplo.** Cálculo de los números de Fibonacci.

  $F(n) = F(n-1) + F(n-2)$

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

  – El cálculo del n-ésimo número de Fibonacci se descompone en calcular los números de Fibonacci n-1 y n-2.

  – *Combinar*: sumar los resultados de los subproblemas.

- La idea de la técnica divide y vencerás es aplicada en muchos campos:

  - Estrategias militares.
  - Demostraciones lógicas y matemáticas.
  - Diseño modular de programas.

Método general

- Requisitos para aplicar divide y vencerás:
  - Necesitamos un **método** (más o menos **directo**) de resolver los problemas de tamaño pequeño.
  - El problema original debe poder dividirse fácilmente en un conjunto de subproblemas, del **mismo tipo** que el problema original pero con una resolución **más sencilla** (menos costosa).
  - Los subproblemas deben ser **disjuntos**: la solución de un subproblema debe obtenerse independientemente de los otros.
  - Es necesario tener un método de **combinar** los resultados de los subproblemas.

Otro Ejemplo:

Multiplicación rápida de matrices

- Supongamos el problema de multiplicar dos matrices cuadradas A, B de tamaños **nxn**. C = AxB

  $C(i, j) = \sum_{k=1..n} A(i, k) \cdot B(k, j)$; Para todo i, j= 1..n

- Método clásico de multiplicación:

```
for i:= 1 to N do
  for j:= 1 to N do
    suma:= 0
    for k:= 1 to N do
      suma:= suma + a[i,k]*b[k,j]
    end
    c[i, j]:= suma
  end
end
```

- El método clásico de multiplicación requiere $\Theta(n^3)$.

Multiplicación rápida de matrices

- Aplicamos **divide y vencerás**:
  Cada matriz de **nxn** es dividida en cuatro submatrices de tamaño (**n**/2)x(**n**/2): A_{ij}, B_{ij} y C_{ij}.

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

$$C_{11} = A_{11}B_{11} + A_{12}B_{21}$$
$$C_{12} = A_{11}B_{12} + A_{12}B_{22}$$
$$C_{21} = A_{21}B_{11} + A_{22}B_{21}$$
$$C_{22} = A_{21}B_{12} + A_{22}B_{22}$$

- Es necesario resolver 8 problemas de tamaño **n**/2.
- La combinación de los resultados requiere un $O(\mathbf{n}^2)$.

$$t(n) = 8 \cdot t(n/2) + a \cdot n^2$$

- Resolviéndolo obtenemos que t(n) es $O(n^3)$.
- Podríamos obtener una mejora si hiciéramos 7 multiplicaciones (o menos)...

Multiplicación rápida de matrices

- El tiempo de ejecución será:

$$t(n) = 7 \cdot t(n/2) + a \cdot n^2$$

- Resolviéndolo, tenemos que:
  $t(n) \in O(n^{\log_2 7}) \approx O(n^{2.807})$.

- Las constantes que multiplican al polinomio son mucho mayores (tenemos muchas sumas y restas), por lo que sólo es mejor cuando la entrada es muy grande (empíricamente, para valores en torno a **n**>120).

Multiplicación rápida de matrices

- **Multiplicación rápida de matrices (Strassen):**

$$
\begin{aligned}
P &= (A_{11}+A_{22})(B_{11}+B_{22}) \\
Q &= (A_{21}+A_{22})\, B_{11} \\
R &= A_{11}\, (B_{12}-B_{22}) \\
S &= A_{22}(B_{21}-B_{11}) \\
T &= (A_{11}+A_{12})B_{22} \\
U &= (A_{21}-A_{11})(B_{11}+B_{12}) \\
V &= (A_{12}-A_{22})(B_{21}+B_{22})
\end{aligned}
\qquad \Bigg\} \qquad
\begin{aligned}
C_{11} &= P + S - T + V \\
C_{12} &= R + T \\
C_{21} &= Q + S \\
C_{22} &= P + R - Q + U
\end{aligned}
$$

- Tenemos 7 subproblemas de la mitad de tamaño.
- ¿Cuánto es el tiempo de ejecución?

1.4. Algoritmos de Programación Dinámica.

Método general

- La base de la **programación dinámica** es el **razonamiento inductivo**: ¿cómo resolver un problema combinando soluciones para problemas más pequeños?
- La idea es la misma que en **divide y vencerás**... pero aplicando una estrategia distinta.
- **Similitud:**
  - Descomposición recursiva del problema.
  - Se obtiene aplicando un razonamiento inductivo.
- **Diferencia:**
  - Divide y vencerás: aplicar directamente la fórmula recursiva (programa recursivo).
  - Programación dinámica: resolver primero los problemas más pequeños, guardando los resultados en una tabla (programa iterativo).

Método general

Métodos ascendentes y descendentes

- **Métodos descendentes (divide y vencerás)**
  - Empezar con el problema original y descomponer recursivamente en problemas de menor tamaño.
  - Partiendo del problema grande descendemos hacia problemas más sencillos.
- **Métodos ascendentes (programación dinámica)**
  - Resolvemos primero los problemas pequeños (guardando las soluciones en una tabla). Después los vamos combinando para resolver los problemas más grandes.
  - Partiendo de los problemas pequeños avanzamos hacia los más grandes.

Método general

- **Ejemplo. Cálculo de los números de Fibonacci.**

$$F(n) = \begin{cases} 1 & \text{Si } n \leq 2 \\ F(n-1) + F(n-2) & \text{Si } n > 2 \end{cases}$$

- **Con divide y vencerás:**

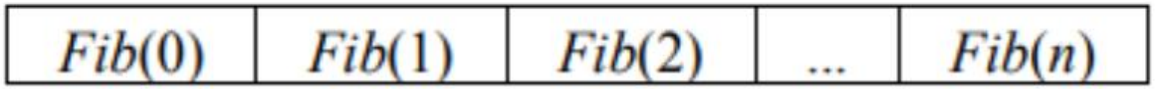
```
operación Fibonacci (n: entero): entero
    si n≤2 entonces devolver 1
    sino devolver Fibonacci(n-1) + Fibonacci(n-2)
```

- **Con programación dinámica:**

```
operación Fibonacci (n: entero): entero
    T[1]:= 1; T[2]:= 1
    para i:= 3, …, n hacer
        T[i]:= T[i-1] + T[i-2]
    devolver T[n]
```

La versión más conocida del algoritmo Fibonacci es poco eficiente ya que su tiempo de ejecución es de orden exponencial. La falta de eficiencia del algoritmo se debe a que se producen llamadas recursivas repetidas para calcular valores de la sucesión, que habiéndose calculado previamente, no se conserva el resultado y es necesario volver a calcular cada vez.

Es posible diseñar un algoritmo que en tiempo lineal lo resuelva mediante la construcción de una tabla que permita ir almacenando los cálculos realizados hasta el momento para reutilizarlos:



El algoritmo iterativo que calcula la sucesión de Fibonacci utilizando tal tabla es:

```
TYPE TABLA = ARRAY [0..n] OF CARDINAL

PROCEDURE FibIter(VAR T:TABLA;n:CARDINAL):CARDINAL;

     VAR i:CARDINAL;

BEGIN
     IF
          n<=1 THEN RETURN 1
     ELSE
          T[0]:=1;
          T[1]:=1:
          FOR i:=2 TO n DO
               T[i]:=T[i-1]+T[i-2]
          END;
          RETURN T[n]
     END
END FibIter;
```

Análisis de tiempos de ejecución

- La programación dinámica se basa en el uso de **tablas** donde se almacenan los resultados parciales.
- En general, el **tiempo** será de la forma:

  tamaño de la tabla * tiempo de rellenar cada elemento de la tabla.

- Un aspecto **importante** es la memoria que puede llegar a ocupar la tabla.
- Además, algunos de estos cálculos pueden ser innecesarios.

---------- *FIN DEL DOCUMENTO*