Grado en Ingeniería Información

Estructura de Datos y Algoritmos

Sesión 14

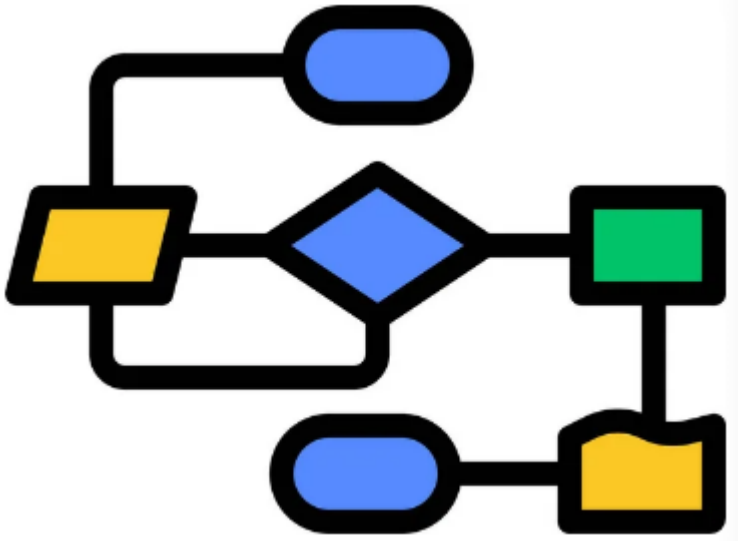
Curso 2022-2023

Marta N. Gómez

UNIVERSIDAD NEBRIJA

T4. Técnicas Algorítmicas

- Divide y Vencerás
- Algoritmos Voraces
- Backtracking

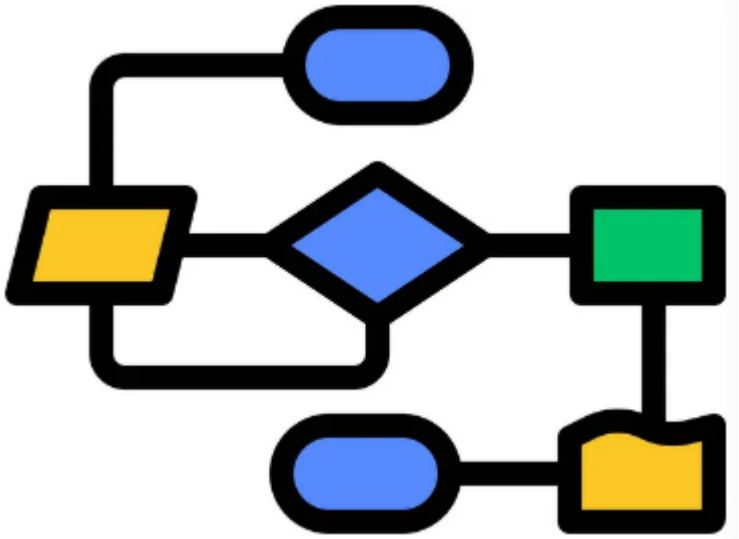


UNIVERSIDAD NEBRIJA

ÍNDICE

T4. Técnicas Algorítmicas

- **Divide y Vencerás**
- Algoritmos Voraces
- Backtracking



UNIVERSIDAD NEBRIJA

ÍNDICE

La técnica ***divide y vencerás*** consiste en:

- **Descomponer** un problema en un conjunto de **subproblemas del mismo tipo**, pero más pequeños.
- La **resolución** de los subproblemas se hace **aplicando la misma técnica**.
- La **combinación de las soluciones** permite resolver el **problema original**.

Normalmente, la resolución de los subproblemas se hace de ***forma recursiva***.

Proceso:

- Si el **problema (P)** tiene **solución directa** para los datos **(D), se resuelve.**
- En caso contrario los pasos son los siguientes:
  1. **Dividir el problema**: se plantea el problema de forma que se pueda descomponer en ***k* subproblemas de igual tipo** y de **menor tamaño**.
  2. **Resolver los subproblemas**: solución de forma independiente, directamente cuando son elementales (caso base) o de forma recursiva.
  3. **Combinar las soluciones**: construir la solución del problema original combinando las soluciones obtenidas anteriormente.

Esquema del algoritmo:

```
tip_Sol DivideyVenceras (tip_Pb pb) {
    if (esCasoBase(pb) {
        return solucionCasoBase(pb);
    }
    else {
        dividirProblema(pb, subPb);
        for (i{0}; i < subPb.size(); i++) {
            Sol_subPb.at(i) = DivideyVenceras(subPb.at(i));
        }
        return combinarSoluciones(Sol_subPb);
    }
}
```

Consideraciones:

- El **número** de **subproblemas** (***k***) debe ser **pequeño**.
- Los **subproblemas** deben tener un **tamaño parecido** y que **no se solapen** entre sí.
- Las operaciones de **dividirProblema y combinarSoluciones** sean bastante eficientes.

Algoritmos de simplificación, cuando **k = 1**, estos algoritmos se pueden implementar de **forma iterativa**, lo que mejora su complejidad. Por ejemplo, el cálculo del factorial.

Eficiencia:

Si el problema ***x*** es de tamaño ***n*** y los subproblemas ***x1, x2, …, xk*** son de tamaño ***n1, n2, …, nk***, respectivamente, el **coste en tiempo** del diseño del algoritmo recursivo de divide y vencerás produce **la ecuación de recurrencia** de la forma:

$$T(n) = \begin{cases} g(n) & n \leq n_0 \\ \sum_{j=1}^{k} T(n_j) + f(n) & n > n_0 \end{cases}$$

donde:

$T(n)$: coste del algoritmo para el problema de tamaño n

n_0: tamaño umbral para no seguir dividiendo

$g(n)$: coste del caso base

$f(n)$: coste de descomponer el problema y combinar soluciones

Eficiencia:

Muchos algoritmos **divide y vencerás** responden a la **ecuación de recurrencia** de la forma:

$$T(n) = \begin{cases} c & si\ 0 \le n < b \\ aT(n/b) + f(n) & si\ n \ge b \end{cases}$$

donde:

- a: número de subproblemas
- **n/b**: tamaño de cada subproblema

Además, suponiendo que $f(n) \in \theta(n^k)$

$a < b^k$	$T(n) \in \theta(n^k)$
$a = b^k$	$T(n) \in \theta(n^k log\ n)$
$a > b^k$	$T(n) \in \theta(n^{log_b a})$

Adivinar un número natural positivo:

Se trata de pensar un número natural positivo y que lo adivinen simplemente preguntando si es menor o igual que otros números.

Algoritmo de búsqueda binaria:

Se compara el **dato a buscar** con el **elemento central del vector**:

- Si es el elemento buscado, se **finaliza**.
- Si no, se sigue **buscando en la mitad del vector** que determine la relación entre el valor del elemento central y el buscado.

El algoritmo finaliza cuando se **localiza el dato** buscado en el vector o **se termina el vector** porque no existe.

EJEMPLO: Divide y Vencerás

Búsqueda Binaria de un elemento de un Vector Ordenado

[0]	[1]	[2]	[3]	[4]	[5]	[6]
1	3	5	7	8	12	15

15

[0]	[1]	[2]	[3]	[4]	[5]	[6]
1	3	5	7	8	12	15

15>7

[0]	[1]	[2]	[3]	[4]	[5]	[6]
1	3	5	7	8	12	15

15>12

[0]	[1]	[2]	[3]	[4]	[5]	[6]
1	3	5	7	8	12	15

15=15

EJEMPLO: Divide y Vencerás

Búsqueda Binaria de un elemento de un Vector Ordenado

Análisis en el mejor caso

```
int busquedaBinaria (const string & f, char l)
{
    int i, ppio{0}, final;

    final = f.size()-1;

    while (ppio <= final)
    {
        i = (ppio+final)/2;

        if (l == f.at(i)) {
            return i;
        }
        else if (l < f.at(i)) {  // se busca en la mitad izquierda
            final = i-1;
        }
        else {                   // se busca en la mitad derecha
            ppio = i+1;
        }
    }

    return -1;
}
```

← $\Omega(1)$

← $\Omega(1)$

← $\Omega(1)$

← $\Omega(1)$

← $\Omega(1)$

$\Omega(1)$

Luego: $T(n)$ es $\Omega(1)$

EJEMPLO: Divide y Vencerás

Búsqueda Binaria de un elemento de un Vector Ordenado

Análisis en el peor caso

```
int busquedaBinaria (const string & f, char l)
{
    int i, ppio{0}, final;                                  ← O(1)

    final = f.size()-1;                                     ← O(1)

    while (ppio <= final)                                   ← O(log n)
    {
        i = (ppio+final)/2;                                 ← O(1)

        if (l == f.at(i)) {                                 ← O(1)
            return i;
        }
        else if (l < f.at(i)) {  // se busca en la mitad izquierda
            final = i-1;                                    ← O(1)
        }
        else {                   // se busca en la mitad derecha
            ppio = i+1;                                     ← O(1)
        }
    }

    return -1;                                              ← O(1)
}
```

O(1) } O(log n) } O(log n)

Luego: *T(n)* es *O(log n)*

EJEMPLO: Divide y Vencerás

Búsqueda Binaria de un elemento de un Vector Ordenado

```
int busquedaBinariaREC (const string &f, char l, int ppio, int final)
{
    int i;

    if (ppio > final) return -1;
    else {
        i = (ppio+final)/2;
        if (l == f.at(i)) {
            return i;
        }
        else if (l < f.at(i)) {
            // se busca en la mitad izquierda
            return busquedaBinariaREC(f, l, ppio, i-1);
        }
        else {
            // se busca en la mitad derecha
            return busquedaBinariaREC(f, l, i+1, final);
        }
    }
}
```

Luego: $T(n)$ es $O(\log n)$

Algoritmo de búsqueda binaria recursivo:

El problema se descompone en un subproblema de tamaño n/2 y el coste de la descomposición y la combinación de soluciones es constante:

$$T(n) = T(n/2) + O(1) =$$
$$T(n/4) + O(1) + O(1) = \ldots = T(n/2^{log^n}) +$$
$$log\ n\ O(1) = T(n/n) + log\ n\ O(1) = 1 +$$
$$log\ n\ O(1) \in O(log\ n)$$

Luego: $T(n)$ es $O(log\ n)$

Solución - Adivinar un número natural positivo:

Se basa en el algoritmo de la **búsqueda binaria**, pero sin vector.

El problema siempre tiene solución.

El algoritmo devuelve el número que está buscando como resultado.

Utiliza una función "***esMenorIgual***" que devuelve true si y solo si en número que hay que adivinar es menor o igual que el que recibe dicha función como parámetro.

EJEMPLO 1: Divide y Vencerás

Adivinar un número natural positivo

```
int busquedaBinariaJuego (int ini, int fin) {
    if (ini == fin) {
        return fin;
    }
    else {
        int mitad = (ini + fin) / 2;
        if (esMenorIgual(mitad)) {
            return busquedaBinariaJuego (ini, mitad);
        }
        else {
            return busquedaBinariaJuego (mitad+1, fin);
        }
    }
}
```

← *O(1)*

O(log n)

EJEMPLO 1: Divide y Vencerás

Adivinar un número natural positivo

```
int adivinarNumero () {
    // Primero hay que encontrar la cota inferior y superior
    int cinf{1}, csuo{1};

    while (!esMenorIgual(csup)) {
        cinf = csup + 1;
        csup = 2 * csup;
    }

    return busquedaBinariaJuego (cinf, csup);
}
```

O(log n)

O(log n)

O(log n)

Emparejar tornillos y tuercas:

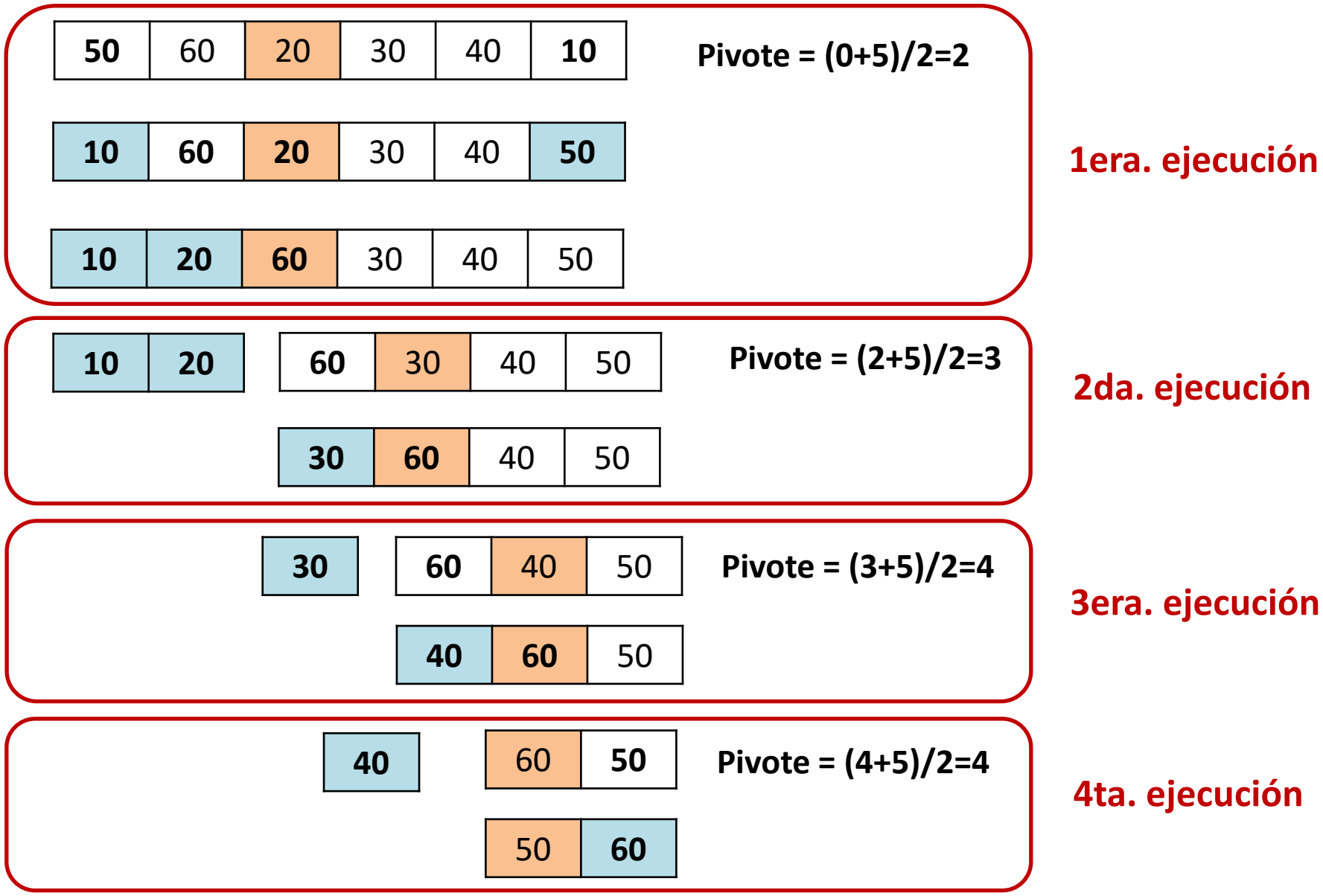
Se tienen dos cajones, uno con ***n tornillos*** de varios tamaños, y otro con sus correspondientes ***n tuercas***. Se necesita **emparejar cada tornillo con su tuerca**, pero en la habitación no se dispone de luz y, por tanto, no es posible realizar la comparación visual. La única posibilidad es tratar de **enroscar una determinada tuerca con un tornillo para si es grande, pequeña o se ajusta bien al tornillo**.

Ordenación rápida (Quicksort)

- El algoritmo consiste en **dividir el vector** que se desea ordenar en **dos bloques**. En el primero se sitúan todos los elementos que son **menores** que un valor que se toma como referencia (**pivote**) y en segundo bloque el resto.
- Este procedimiento se repite dividiendo a su vez cada uno de estos bloques y repitiendo la operación anteriormente descrita.
- La condición de parada se da cuando el bloque que se desea ordenar está formado por un **único elemento** (bloque ordenado).
- El resultado se obtiene de la combinación de todos los resultados parciales.

EJEMPLO: Divide y Vencerás

Ordenación rápida (Quicksort)



EJEMPLO: Divide y Vencerás

Ordenación rápida (Quicksort)

```
void ordenQuickSort(vector <int> & v, int izq, int der)
{
    int i{izq}, d{der}, pivote,aux;

    pivote = v.at((i+d)/2);

    while (i < d)
    {
       while (v.at(i) < pivote) { i++;  }

       while (pivote < v.at(d)) { d--;  }

       if (i <= d)  // intercambio de elementos
       {    aux = v.at(i);
            v.at(i) = v.at(d);
            v.at(d) = aux;
            i++;  d--; // se ajustan las posiciones
       }
    }

    if (izq < d) {  ordenQuickSort(v, izq, d);   }
    if (i < der) {  ordenQuickSort(v, i, der);   }
}
```

Análisis del mejor caso: Se produce cuando el pivote divide al vector en dos partes iguales y el orden de los elementos es aleatorio:

$$T(n) = 2T\left(\frac{n}{2}\right) + n - 1$$

$T(n/2)$ es el coste de ordenar una de las mitades y n el número de comparaciones realizadas.

Si desarrollamos la recurrencia se obtiene:

$$T(n) = n\,logn - n - 1$$

Que es de orden $T(n) \in \Omega(n\,logn)$

Análisis del mejor caso: Supongamos que el tamaño del vector es una potencia de 2: $n = 2^k \rightarrow log\ n = k$

Recorrido	Comparaciones
1º	(n-1) comparaciones * 1 vector 2 vectores de tamaño n/2
2º	(n/2) comparaciones * 2 vector = n 4 vectores de tamaño n/4
3º	(n/4) comparaciones * 4 vector = n 8 vectores de tamaño n/8
…	…
k-ésimo	$(n/2^{k-1})$ comparaciones * 2^{k-1} vector = n 2^k vectores de tamaño $n/2^k$

$$n + n + \cdots + n = kn = n\ logn$$

Luego es de orden $T(n) \in \Omega(n\ logn)$

Análisis del peor caso: Si se elige como pivote el primer elemento del vector y además se considera que el vector esta ordenado decrecientemente entonces, el bucle se ejecutará en total: $(n-1)+(n-2)+(n-3)+\cdots+1$

Cada miembro de este sumando proviene de cada una de las sucesivas ordenaciones recursivas. Este sumatorio da lugar a la siguiente expresión:

$$(n-1)+(n-2)+(n-3)+\cdots+1=$$

$$\sum_{i=1}^{n-1}(n-i)=\frac{[(n-1)+1](n-1)}{2}=\frac{n(n-1)}{2}$$

Que es de orden cuadrático $T(n) \in O(n^2)$

Solución - Emparejar tornillos y tuercas:

Si el número de tuercas es **n ≤ 1, caso base**.

Si **n > 1**:

- Se toma **un tornillo cualquiera** y, comparando con él, se divide el conjunto de **tuercas en las menores**, **las mayores y las que enroscan (igual tamaño)**.
- Utilizando **una tuerca de las que enroscan**, se divide el conjunto de **tornillos en menores, mayores y los que enroscan (igual tamaño)**.
- Se hacen las **llamadas recursivas con los conjuntos de tuercas y tornillos menores y mayores**.

Importante: la partición solo es correcta en el caso de que **todos los tronillos y tuercas sean de tamaños diferentes**, porque de otro modo puede que los tamaños de los conjuntos no coincidan

EJEMPLO 2: Divide y Vencerás

Emparejar tornillos y tuercas

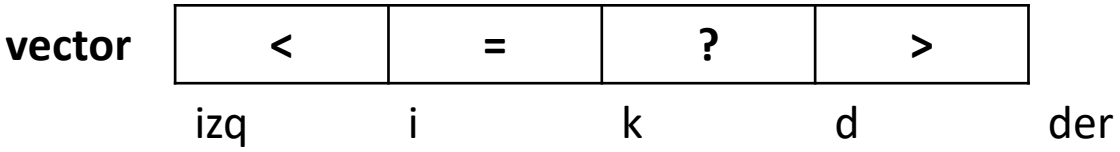
```
// Telem será el tamaño de cada tornillo o tuerca
// Los índices izq y der son valores de los límites de cada partición
void emparejarTornilloTuerca (vector <Telem> &tornillos,
                              vector <Telem> &tuercas, int izq, int der)
{   int i, d, k, l;

    if (izq < der)
    {
       particionVector(tuercas, izq, der, tornillos.at(izq), i, d);
       particionVector(tornillos, izq, der, tuercas.at(i), k, l);

       // Los índice i == k y d == l
       emparejarTornilloTuerca(tornillos, tuercas, izq, i-1);   }
       emparejarTornilloTuerca(tornillos, tuercas, d+1, der);   }
}
```

EJEMPLO 2: Divide y Vencerás

Emparejar tornillos y tuercas



Los índices ***i***, ***k*** se mueven **hacia la derecha**, mientras el índice ***d*** se mueve **hacia la izquierda**.

En cada paso, el algoritmo compara ***V.at(k)*** con el elemento **pivote**:

- Si ***V.at(k) es menor***, se **intercambia con V.at(i) para que se agrupe con los menores**. Además, con el intercambio, en V.at(k) se coloca un elemento igual al pivote. Luego, se intercambian los índices i y k.
- Si **V.at(k) es igual**, no hay cambios.
- Si **V.at(k) es mayor**, se **intercambia con V.at(d) para que se agrupe con los mayores**. Ahora, con el intercambio, en V.at(k) se coloca un elemento desconocido. Luego, solo se puede avanzar, decrementando el índice ***d***.

El bucle termina cuando k > d, es decir, ya no hay elementos mal colocados.

EJEMPLO 2: Divide y Vencerás

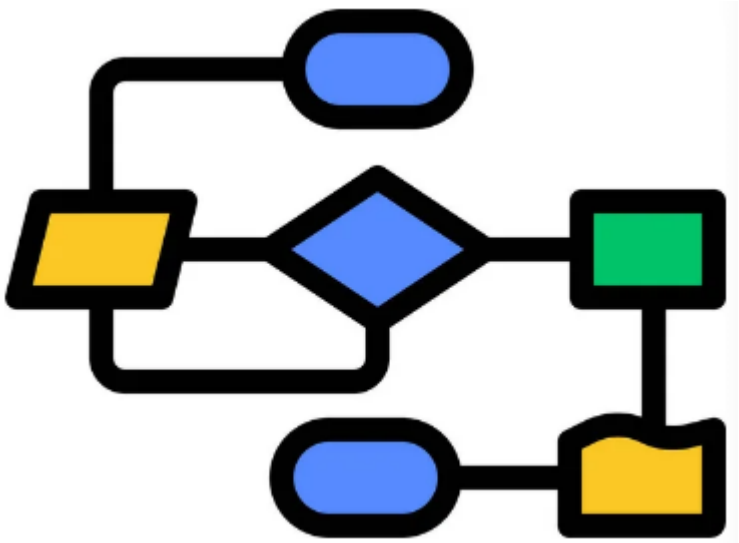
Emparejar tornillos y tuercas

```
// Telem será un tornillo o una tuerca concreta
// Los índices i y j son valores de salida para definir los límites de la partición
void particionVector (vector <Telem> &V, int izq, int der, Telem pivote, int &i, int &d) {
    int k{izq};
    Telem aux;
    i = izq; d = der;

     while (k <= d)
     {  // posible intercambio de elementos y ajuste de posiciones
        if (V.at(k) < pivote) {
            aux = V.at(k);
            V.at(k) = V.at(i);
            V.at(i) = aux;
            i++;  k++;
        } else if (V.at(k) == pivote) {
            k++;
        } if (V.at(k) > pivote) {
            aux = V.at(k);
            V.at(k) = V.at(d);
            V.at(d) = aux;
            d--;
        }
     }
}
```

T4. Técnicas Algorítmicas

- Divide y Vencerás
- **Algoritmos Voraces**
- Backtracking



UNIVERSIDAD NEBRIJA

ÍNDICE

Algoritmos Voraces

Los **algoritmos Voraces (Greedy o ávidos)** describen muy bien lo que hacen:

- Cada etapa se consume una parte de los datos.
- Su objetivo es que, bajo ciertas condiciones, la parte consumida sea lo mayor posible.

Los algoritmos **Voraces** (greedy o ávidos) son básicamente iterativos y tienen una serie de etapas.

Cada **etapa** **consume** una **parte de los datos o candidatos (C)** y **construye** una **parte de la solución**.

Finalizan cuando se han consumido todos los datos o candidatos o se alcanza una solución.

Cada parte consumida se evalúa una única vez, siendo **descartada** (D) o **seleccionada** (S):

- Si es **seleccionada**, forma **parte de la solución**.
- Mientras que si es **descartada**, **no forma parte de la solución ni volverá a ser considerada para la misma**.

Algoritmos Voraces

Los algoritmos **Voraces** (greedy o ávidos) funcionan de la siguiente forma:

1. **Inicialmente** se parte de una solución vacía (S).
2. En **cada etapa**, se analiza en conjunto de candidatos (C) para elegir el elemento que se añade a la solución.
3. Se **termina** cuando el conjunto de elementos seleccionados constituyen una solución.

ESQUEMA: Algoritmos Voraces

```
cjtoSolucion algVoraz (cjtoCandidatos &C)  }
    cjtoSolución S = ∅;
    while (!esSolucion(S) && C ≠ ∅) {
      x = seleccionar(C);
      C = C – {x}
      if (esFactible (S, x))
            insertar(S, x);
    }
    if (esSolucion(S))
      return S;
    else
      return “No se encontró la solución”;
}
```

Funciones:

esSolucion(S): Comprueba si el conjunto de candidatos seleccionado hasta el momento es una solución (independientemente de que sea óptima o no).

Seleccionar(C): Selecciona el elemento más adecuado del conjunto de candidatos pendientes (no seleccionados ni rechazados).

esFactible(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 S. También puede ser necesario hacer otras operaciones.

El coste de los algoritmos **Voraces** depende de:

1. El número de iteraciones del bucle, que depende del tamaño de la solución construida y del tamaño del conjunto de candidatos.
2. El coste de las funciones selección y factible:
   a) La función factible suele tener un coste constante.
   b) La función selección tiene que explorar el conjunto de candidatos.

Algoritmos Voraces

Los algoritmos **Voraces** son bastante eficientes: O(n log n), $O(n^2)$.

No garantizan que la solución sea la óptima del problema, ni siquiera que se obtenga una solución, aunque exista.

EJEMPLO 1: Algoritmos Voraces

Descomposición en factores primos

Descomposición en factores primos:

Sea un número natural n, sus factores primos: $300 = 2^2\, 3^1 5^2$

Conjunto de candidatos: todos los factores primos posibles para el número (menores que su mitad).

Conjunto Solución: todos los factores que dividen al número.

Función objetivo: obtener el cociente 1.

EJEMPLO 1: Algoritmos Voraces

Descomposición en factores primos

Sea un número natural n, sus factores primos: $300 = 2^2\, 3^1 5^2$

Proceso de resolución: eliminar en cada etapa un divisor de n cuantas veces sea posible:

1. Se elimina el 2 tantas veces como sea posible, y se considera el cociente final,
2. Se elimina el 3, de igual manera.
3. Se continua de igual forma con el resto de factores hasta que el cociente sea 1.

EJEMPLO 1: Algoritmos Voraces

Descomposición en factores primos

```
vector<termino> calcularFactores (int N) {
    vector<termino> factores;
    termino ter;
    for (int i{2}; N > 1; i++) {
        ter.veces=0;
        ter.f = i;
        while (N % i == 0) {
            ter.veces++;
            N /=i;
        }
        if (ter.veces > 0) {
            factores.push_back(ter);
        }
    }

    return factores;
}
```

EJEMPLO 2: Algoritmos Voraces

Dar cambio con el menor número de billetes/monedas

Dar cambio con el menor número de billetes/monedas:

Se pide crear un algoritmo que permita a una máquina expendedora devolver el cambio mediante el menor número de billetes posible, considerando que el número de billetes es limitado, es decir, se tiene un número concreto de billetes de cada tipo".

La estrategia a seguir consiste en seleccionar los billetes/monedas de mayor valor que no superen la cantidad de cambio a devolver, en cada etapa.

EJEMPLO 2: Algoritmos Voraces

Dar cambio con el menor número de billetes/monedas

Conjunto de candidatos: todos los tipos de monedas disponibles, suponiendo que de cada tipo hay una cantidad limitada.

Conjunto Solución: las monedas que suman el importe.

Función objetivo: minimizar el número de monedas utilizadas ($\sum_i x_i$).

$$\sum_{i=1..n} x_i\, c_i \qquad x_i \geq 0$$

EJEMPLO 2: Algoritmos Voraces

Dar cambio con el menor número de billetes/monedas

Supongamos que hay que devolver 128 euros y se tiene la siguiente disponibilidad de billetes y monedas:

- 3 billetes de 50 euros
- 2 billetes de 20 euros
- 1 billete de 5 euros
- 6 monedas de 1 euro

1. Se divide 128 entre 50 y se obtiene como cociente 2. Esto representa el número de billetes de 50, quedando una cantidad de cambio a devolver de 28 euros.
2. Se divide 28 entre 20 y se obtiene como cociente 1, que es el número de billetes de 20 que se pueden utilizar sin pasarse. Ahora queda pendiente la cantidad de 8 euros.
3. Se divide 8 entre 5 y se obtiene 1, luego seleccionamos 1 billete de 5.
4. La cantidad de cambio a devolver ahora es 3 euros.
5. Se divide 3 entre 1 y se obtiene como cociente 3, que son las monedas de 1 euro necesarias para terminar el problema de forma correcta.

EJEMPLO 2: Algoritmos Voraces

Dar cambio con el menor número de billetes/monedas

Supongamos que hay que devolver 128 euros y se tiene la siguiente disponibilidad de billetes y monedas:

```
array <int, 4> obtenerCambio (int  T, array <int, 4> &C)  {
    array <int, 4>  S{0, 0, 0, 0};
    int cambio{0}, i{0};
    while (cambio != T) {
        while (C.at(i) > (T – cambio) && i < 4) i++;
        if (i==4) { cout << “No existe solución”;  }
        else {
            S.at(i):= (T - cambio) / C.at(i);
            cambio = cambio + C.at(i) * S.at(i);
        }
    }
    return S;
}
```

EJEMPLO 2: Algoritmos Voraces

Dar cambio con el menor número de billetes/monedas

- Si tenemos 5 monedas de valores C={50, 25, 20, 5, 1} y se tiene que obtener 42, la solución que se obtendría S = {25, 5, 5, 5, 1, 1}, mientras que la solución óptima sería S = {20, 20, 1, 1}.
- Además, si no se incluye la unidad, tampoco se garantiza la solución.
- Para obtener una solución óptima es necesario que el conjunto de candidatos esté formado por valores que sean potencia de un tipo básico. Por ejemplo, C={125, 25, 5, 1}. Así, sólo hay que encontrar la descomposición de la cantidad en base a ese valor, que es única y mínima.

EJEMPLO 2: Algoritmos Voraces

Dar cambio con el menor número de billetes/monedas

- Para que la función de selección funcione de forma adecuada, el vector de valores de billetes/monedas debe estar ordenado en orden decreciente.
- El coste del algoritmo es de O(max(n log n, m) donde n es el número de valores de billetes/monedas de C, y m el número de iteraciones del bucle exterior.

EJEMPLO 3: Algoritmos Voraces

Algoritmo de PRIM