UADE

UADE

Programación III – Diseño y Análisis de Algoritmos

Temas a desarrollar

1	Introducción a la Materia
2	Conceptos Básicos
3	Complejidad Temporal Alg. Recursivos
4	Ejercicios Complejidad Temporal
5	Divide y Conquista
6	Búsqueda Binaria
7	Palabra Palíndromo

Introducción a la Materia

UADE

Acerca de la Cursada

- Modalidad Presencial 100% (con una clase el 10/05 por teams debido a los cambios en el turno noche donde la clase finaliza a las 22hrs., que es sábado de 9 a 13hrs).

- En cada clase una parte se destinará a la explicación teórica y la otra a la resolución de ejercicios por parte de los alumnos, con el objetivo de incentivar una participación activa de los mismos.

- Durante la resolución de los ejercicios los alumnos podrán trabajar en forma individual o grupal, se aconseja ésta segunda modalidad para poder un intercambio de ideas.

- Se solicitará a los alumnos que pase al pizarrón a mostrar la resolución de los ejercicios realizados.

Bibliografía

BRASSARD, Gilles. *Fundamentals of algorithmics.* Upper Saddle River: Prentice Hall, 1996. ISBN: 9780133350685

Información de Aprobación y Fechas Importantes

- Aprobación: se debe aprobar el parcial con 4 o más, con una instancia de recuperatorio, y un TPO obligatorio en grupo con evaluación individual y nota mayor a 4.
- Parcial: **14/05/2025**
- Entrega TPO: **11/06/2025 y 18/06/2025**
- Defensa TPO: **25/06/2025**
- Recup./Final Reg. Optativo: **02/07/2025**
- Final Regular: **23/07/2025**

Qué es Programación III – Diseño y Análisis de Algoritmo

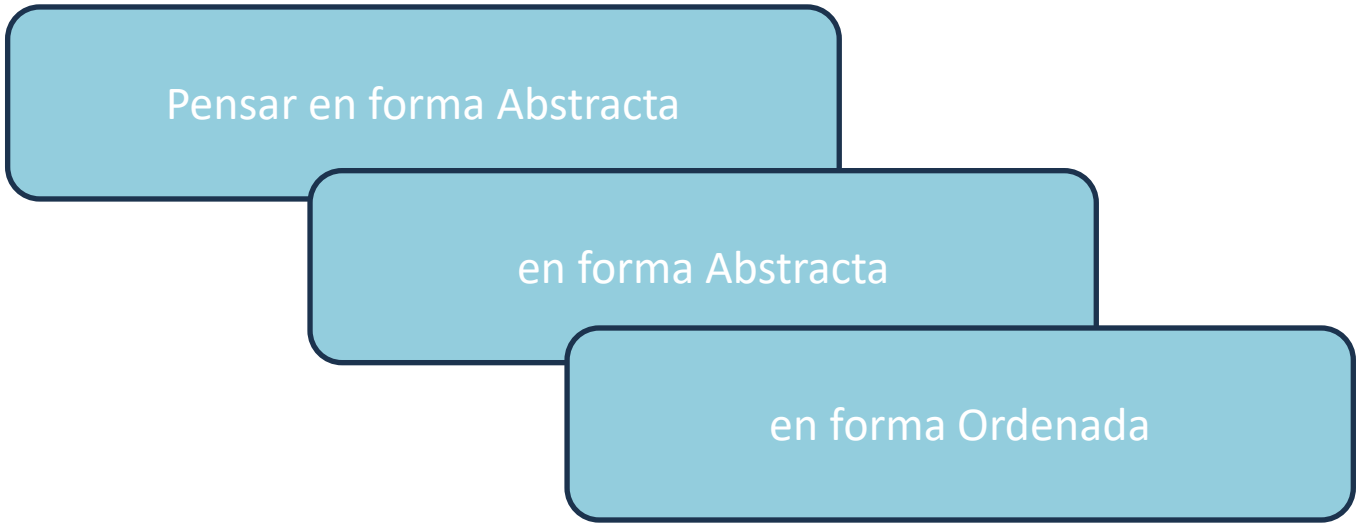
- Estudio de diferentes técnicas de diseño de algoritmos, con el objetivo de encontrar la solución más adecuada desde el punto de vista de eficiencia temporal a distintos problemas.
- Se van a estudiar cuatro técnicas principales, Divide y Conquista, Greedy, Programación Dinámica y Backtracking (incluyendo diferentes conceptos asociados).

Por qué Algoritmos

- Los algoritmos son independientes de los lenguajes de programación que se utilicen para codificar la solución diseñada, entendiendo que la estrategia definida puede aplicarse luego en el lenguaje de programación que se desee.
- El análisis de la eficiencia temporal teórica es uno de los puntos más importantes para evaluar el algoritmo diseñado, previamente a su programación.

En qué nos basamos para diseñar un algoritmo

- Debemos tener en cuenta lo siguiente:



Por qué vamos a trabajar en pseudocódigo

- El pseudocódigo nos permite

Abstraernos del lenguaje de programación

Se aproxima más al lenguaje natural

Aunque nos permite usar las sentencias similares a los lenguajes de programación.

Conceptos básicos de algoritmos

Conocimientos Esperados

Se espera que el alumno conozca los siguientes conceptos de las materias previamente cursadas:

- Se trabajaran con tipos de datos primitivos (enteros, char, decimal, etc.)
- Se trabajaran con estructuras de acceso directo (arreglos, vectores, etc.).
- Se espera que conozcan recorridos sobre vectores, matrices, etc.
- Se espera que el alumno tenga conocimientos matemáticos básicos de matrices, funciones de una, dos o más incógnitas y generación de gráficos de dos dimensiones.

Conceptos básicos – Programa/Instancia

- **Programa**: es la escritura de la estrategia de un algoritmo en un lenguaje de programación en particular (Java, C++, Delphi, etc.) para ser ejecutado en una computadora.

- **Instancia**: una instancia de un problema es una combinación válida de los posibles datos de entrada. Un problema puede tener una o más instancias, incluso infinitas.

- Un algoritmo será correcto o eficaz si funciona de la forma esperada para todas las instancias de un problema.

Conceptos básicos – Eficiencia/Correctitud

- Un algoritmo será **eficiente** si resuelve un problema utilizando una cantidad acotada de recursos y tiempo.

- La algoritmia se ocupa, entonces, del estudio de los algoritmos y abarca el análisis de la **correctitud** y de la **eficiencia.** Es decir, analiza si un algoritmo resuelve un problema y la cantidad de recursos necesarios para llegar a la solución.

Conceptos básicos – Eficiencia temporal

- **Medición práctica**: a través de la medición del tiempo que lleva obtener una solución para una entrada de datos dada. Para poder llevar a cabo esta medición se requiere tener un programa del algoritmo.

- **Medición teórica**: se trata de medir la cantidad de operaciones elementales que ejecuta para un tamaño de datos de entrada.

- La eficiencia nos permite comparar diferentes algoritmos/programas entre sí en cuanto a los usos de recursos.

- No debe depender del *hardware* ni del lenguaje de programación.

Conceptos básicos – Eficiencia temporal

- **Principio de invarianza**

> *Dado un algoritmo A y dos implementaciones del mismo I_1 e I_2, donde la primera tarda $T_1(n)$ y la segunda $T_2(n)$ segundos, entonces existen un número real $c > 0$ y un entero n_0 tal que $T_1(n) \leq cT_2(n)$ para todo $n > n_0$.*

- Esto significa que dos implementaciones distintas de un mismo algoritmo no difieren entre sí en más de una constante multiplicativa.

Cálculo eficiencia

- **Cálculo del número de operaciones elementales**

  - Se considera **operación elemental constante** a las asignaciones, operaciones aritméticas o lógicas, acceso a la posición de un vector, llamada a un método (en este caso se deberá analizar la complejidad del método). En todos estos casos suma una constante **1 (UNO).**

  - Para **una secuencia de operaciones**, la cantidad de operaciones elementales es la suma de la cantidad de operaciones elementales de cada parte.

Cálculo eficiencia

- **Cálculo del número de operaciones elementales**

  - Para un ***condicional***, la cantidad de operaciones elementales es el costo de la evaluación de la condición más el máximo entre la cantidad de operaciones de las dos ramas de la condición.

  - Para un ***ciclo***, la cantidad de operaciones elementales es N veces la cantidad de operaciones elementales del bloque de repetición, donde N es la cantidad de veces que se repite el ciclo.

  - Para una **llamada a un método**, el número de operaciones elementales es 1 más el número de operaciones elementales de la ejecución del método.

Cálculo eficiencia – tipos de análisis

- **Peor caso**: es el mayor de los costos de todas las instancias de un problema.
- **Caso promedio**: es el promedio de los costos de todas las instancias de un problema.
- **Mejor caso**: es el menor de los costos de todas las instancias de un problema.

Nos centraremos en el análisis del peor caso ya que es el que aporta una cota superior de todos los demás casos.

Notación asintótica

$f(n) = 1$	constante
$f(n) = \log(n)$	logarítmica
$f(n) = n$	lineal
$f(n) = n \log(n)$	
$f(n) = n^2$	cuadrática
$f(n) = n^k$	polinomial
$f(n) = 2^n$	exponencial
$f(n) = b^n$	exponencial
$f(n) = n!$	factorial

Complejidad en algoritmos recursivos

Resolución de recurrencia

- Resolución de funciones/procedimientos recursivos: cálculo de la función factorial de n.

```
Factorial
Entrada: entero n
Salida: entero s valor factorial
    si (n = 1)
        devolver 1
    fin si
     sino
         devolver n*factorial (n-1)
     fin sino
Fin algoritmo factorial
```

Resolución de recurrencia - Métodos

- Sustracción: la llamada recursiva es del tipo T(n-b), donde la función es del tipo

$$T(n) = \begin{cases} c & \text{si } 0 \leq n < b \\ aT(n-b) + p(n) & \text{si } n \geq b \end{cases}$$

Donde *a* es la cantidad de llamadas recursivas en el peor de los casos. *b* es la cantidad de unidades en que disminuye la entrada en cada llamado recursivo. *k* es el grado del polinomio *p(n)* de aquellas sentencias a ejecutar fuera del llamado recursivo. Entonces se llega a:

$$T(n) \in \begin{cases} \Theta(n^k) & \text{si } a < 1 \\ \Theta(n^{k+1}) & \text{si } a = 1 \\ \Theta(a^{n \; div \; b}) & \text{si } a > 1 \end{cases}$$

Solución sustracción

```
Factorial
Entrada: entero n
Salida: entero s valor factorial
    si (n = 1)
        devolver 1
    fin si
     sino
         devolver n*factorial (n-1)
     fin sino
Fin algoritmo factorial
```

El factorial es del tipo sustracción T (n-b), dónde a=1, b=1 y k=0

$$T(n) \in \begin{cases} \Theta(n^k) & \text{si } a < 1 \\ \Theta(n^{k+1}) & \text{si } a = 1 \\ \Theta(a^{n \ div \ b}) & \text{si } a > 1 \end{cases}$$

Como a=1 se da la opción $\Theta(n^{k+1})$, y al ser k=0, la complejidad es:

$\Theta(n)$

Resolución de recurrencia - Métodos

- División: la llamada recursiva es del tipo T(n/b), donde la función es del tipo

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

Donde *a* es la cantidad de llamadas recursivas en el peor de los casos. *b* es la cantidad de unidades en que se divide la entrada en cada llamado recursivo. *k* es el grado del polinomio *f(n)* de aquellas sentencias a ejecutar fuera del llamado recursivo. Entonces se llega a:

$$T(n) \in \begin{cases} \Theta(n^k) & \text{si } a < b^k \\ \Theta(n^k \log(n)) & \text{si } a = b^k \\ \Theta(n^{\log_b(a)}) & \text{si } a > b^k \end{cases}$$

Caso División

```
BuscarValor
Entrada: árbol AVL a, entero n
Salida: verdadero o falso
    si (a es vacío)
        devolver falso
    sino
        si (a.raiz = n)
            devolver verdadero
        fin si
        sino
            si a.raíz < n
                devolver buscarValor(a.hijoDer, n)
            fin si
            sino
                devolver buscarValor(a.hijoIzq, n)
            fin sino
        fin sino
    fin sino
Fin algoritmo buscarValor
```

BuscarValor es del tipo división T (n/b), dónde a=1, b=2 y k=0

$$T(n) \in \begin{cases} \Theta(n^k) & \text{si } a < b^k \\ \Theta(n^k \log(n)) & \text{si } a = b^k \\ \Theta(n^{\log_b(a)}) & \text{si } a > b^k \end{cases}$$

Como $a=b^k$ la opción es $\Theta(n^k \log(n))$, como $k=0$:

$$\Theta(\log(n))$$

Donde n depende de la cantidad de valores en el árbol

Ejercicios a Resolver

Cálculos de la complejidad temporal

- Complejidad temporal: funciones recursivas

$$T(n) = \begin{cases} c & \text{si } 0 \leq n < b \\ aT(n-b) + p(n) & \text{si } n \geq b \end{cases}$$

$$T(n) \in \begin{cases} \Theta(n^k) & \text{si } a < 1 \\ \Theta(n^{k+1}) & \text{si } a = 1 \\ \Theta(a^{n \; div \; b}) & \text{si } a > 1 \end{cases}$$

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

$$T(n) \in \begin{cases} \Theta(n^k) & \text{si } a < b^k \\ \Theta(n^k \log(n)) & \text{si } a = b^k \\ \Theta(n^{\log_b(a)}) & \text{si } a > b^k \end{cases}$$

Ejercicios de aplicabilidad de complejidad temporal

```
ALGORITMO   AlgoritmoA
Entrada: n
        bool a = obtenerValor(n)
        si (n = 1)
                devolver a
        fin si
        sino
                si (a = verdadero)
                        imprimir (n)
                        AlgoritmoA(n/2)
                fin si
                sino
                        AlgoritmoA(n/2)
                fin sino
         fin sino
fin Algoritmo A
```

Caso división

a= 1

b= 2

P($O(n^2)$)→k = 2

$a ? b^K$ → $a ? b^2$ 1 < 4 →$O(n^K)$ → $O(n^2)$

$$T(n) \in \begin{cases} \Theta(n^k) & \text{si } a < b^k \\ \Theta(n^k \log(n)) & \text{si } a = b^k \\ \Theta(n^{\log_b(a)}) & \text{si } a > b^k \end{cases}$$

Nota: la complejidad de la función “obtenerValor” es $O(n^2)$ e imprimir O(n)

Ejercicios de aplicabilidad de complejidad temporal

```
ALGORITMO   AlgoritmoB
Entrada: n
        bool a = obtenerValor(n)
        si (n = 1)
                devolver a
        fin si
        sino
                si (a = verdadero)
                        calcularCosto (n)
                        AlgoritmoB(n-2)
                fin si
                sino
                        AlgoritmoB(n-2)
                fin sino
        fin sino
fin Algoritmo B
```

nota: la complejidad de la función "obtenerValor" es $\Theta(C)$ y "calcularCosto" es $\Theta(n^2)$

Caso resta

a= 1

b= 2

$P(O(n^2)) \rightarrow k = 2$

a ? 1→ 1=1 →$O(n^{K+1})$ → $O(n^{2+1})$ → $O(n^3)$

$$T(n) \in \begin{cases} \Theta(n^k) & \text{si } a < 1 \\ \Theta(n^{k+1}) & \text{si } a = 1 \\ \Theta(a^{n \; div \; b}) & \text{si } a > 1 \end{cases}$$

Ejercicios de aplicabilidad de complejidad temporal

```
ALGORITMO   AlgoritmoC
Entrada: n
        bool a = obtenerValor(n)
        si (n = 1)
                devolver a
        fin si
        sino
                entero c = 1
                mientras (c <= 3)
                        AlgoritmoC(n-2)
                        c ← c+1
                fin mientras
        fin sino
fin Algoritmo C
```

nota: la complejidad de la función "obtenerValor" es Θ(C)

Caso Resta

a= 3 porque itera el ciclo en donde esta el llamado recursivo 3 veces

b= 2

P(O(C))→k = 0

a ? 1→ 3>1 →$O(a^{n \text{ div } b})$ → $O(3^{n \text{ div } 2})$ → $O(3^{n/2})$ complejidad exponencial

$$T(n) \in \begin{cases} \Theta(n^k) & \text{si } a < 1 \\ \Theta(n^{k+1}) & \text{si } a = 1 \\ \Theta(a^{n\ div\ b}) & \text{si } a > 1 \end{cases}$$

Divide y Conquista

UADE

Divide y Conquista

- **Divide y Conquista** es una técnica de diseño de algoritmos que consiste en dividir un problema "*grande*" en problemas más "*pequeños*", de resolución más sencilla, y luego combinar las soluciones de los problemas más "*pequeños*" para hallar la solución del problema "*grande*".

- El problema original se divide en problemas más pequeños de la misma naturaleza hasta llegar a un problema lo suficientemente "*pequeño*" de resolución en "**forma trivial**". Ésta división se va realizando en forma recursiva.

Divide y Conquista – Estrategia de Solución

Casos bases o triviales	Son aquellos casos que el tamaño de problema tiene solución directa. Podría ser que el caso base sea solo uno.
Subproblemas del mismo tipo	Se debe identificar cómo se va a ir particionando la entrada en tamaños menores, para ir resolviendo cada unos de esos subproblemas por separado en forma independiente.
Combinación de subproblemas	Definir cómo se unificaran los resultados de los subproblemas para dar la solución al problema de tamaño n, es decir al original para el cual se estaba buscando la solución

Divide y Conquista – Esquema General de Solución

```
ALGORITMO DyC
Entrada: x
  si esPequeno(x)
    devolver solucion_trivial(x)
  sino
    ⟨x_1, ..., x_n⟩ ← descomponer(x)
    para i = 1 hasta n
      y_i ← DyC(x_i)
    fin para
    devolver combinar(y_1, ..., y_n)
  fin si
```

Nota: se debe entender que el esquema anterior es genérico y en cada problema pueden no aparecer todos los métodos descriptos.

Problema Búsqueda Binaria

Búsqueda Binaria

El problema de la búsqueda binaria consiste en buscar un número **x** dado en una secuencia ordenada en forma creciente de números **S** que también es un dato de entrada.

Como resultado del problema se deberá devolver "*verdadero*" en el caso de que el **x** pertenezca a **S**, y "*falso*" en caso contrario.

Búsqueda Binaria – Estrategia de Resolución

Dada la naturaleza del problema se puede diseñar una resolución por Divide y Conquista siguiendo la siguiente estrategia:

Caso Base

1. Si la longitud de **S** es UNO se puede determinar unívocamente si **x** pertenece a **S** o no.
2. Si tomamos el valor ubicado a la mitad de **S**, verificamos si es igual a **x** ya tenemos el resultado.

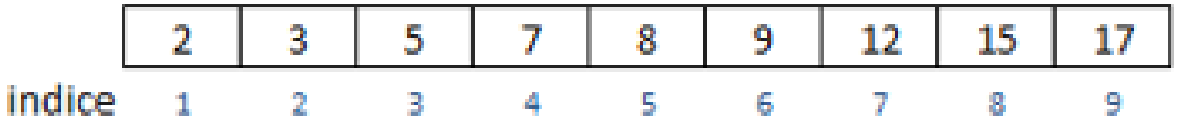
Subproblemas

1. Si x es mayor al valor ubicado en la mitad de S buscamos en la mitad derecha ubicada entre (n/2)+1 y n
2. Si x es menor al valor ubicado en la mitad de S buscamos en la mitad izquierda ubicada entre 1 y (n/2)-1

Combinación

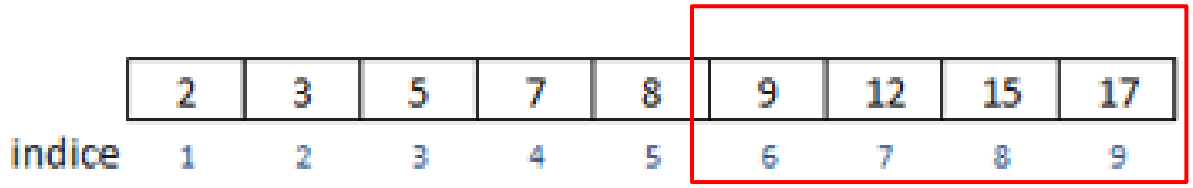
1. En este caso en particular se deberá devolver en forma directa el resultado del subproblema evaluado dado que solo uno de ellos se ejecutará.

Búsqueda Binaria – Ejemplo



x = 12

Si obtenemos la mitad, seria (9+1)/2= 5. Si evaluamos el valor en la posición 5 vemos que es 8, por lo cual no es igual al 12 que buscamos. Como ese valor de la mitad es menor a 12 se busca hacia la parte derecha:



x = 12

Si obtenemos la mitad a partir de la parte derecha considerada que iría entre los índices 6 y 9, seria (9+6)/2= 7 (considerando la parte entera inferior). Si evaluamos el valor en la posición 7 vemos que es 12, por lo cual es igual al 12 que buscamos y ya podríamos devolver "*verdadero*".

Búsqueda Binaria – Algoritmo

```
ALGORITMO BUSQUEDABIN
Entrada: S: Vector<entero>, x: entero
Salida: verdadero o falso
  si longitud(S) = 1
    devolver  S[0] = x
  sino
    y ← S[longitud(S)/2]
    si x = y
      devolver  verdadero
    sino
      mitad ← longitud(S)/2
      si x < y
        devolver  BusquedaBin(S[0, mitad − 1], x)
      sino
        devolver  BusquedaBin(S[mitad ,longitud(S) − 1], x)
      fin si
    fin si
  fin si
```

Búsqueda Binaria – Complejidad Temporal

ALGORITMO BUSQUEDABIN
Entrada: S: Vector<entero>, x: entero
Salida: verdadero o falso
  si $longitud(S) = 1$
    devolver $S[0] = x$
  sino
    $y \leftarrow S[longitud(S)/2]$
    si $x = y$
      devolver verdadero
    sino
      $mitad \leftarrow longitud(S)/2$
      si $x < y$
        devolver $BusquedaBin(S[0, mitad - 1], x)$
      sino
        devolver $BusquedaBin(S[mitad, longitud(S) - 1], x)$
      fin si
    fin si
  fin si

$$T(n) = \begin{cases} c & \text{si } n = 1 \\ 1T(n/2) + c & \text{si } n > 1 \end{cases}$$

$a = 1,\ b = 2$ y $k = 0$, entonces

$$\Theta(n^0 \log(n) = \Theta(\log(n))$$

UADE

Problema de una palabra Palíndromo

Determinar si una palabra Palíndromo

- Una palabra se dice que es Palíndromo si se lee igual en un sentido que en otro, es decir, si se mira de izquierda a derecha es igual que de derecha a izquierda. Si se trata de números en lugar de letras, se llama capicúa.

- Ejemplos de palabras que cumplen la definición de Palíndromo son:

  Ana, radar, reconocer, rotor, salas, somos, etc.

- Lo que se requiere es una estrategia de solución que a partir de una palabra dada determine si la misma es Palíndromo o no.

Determinar si una palabra Palíndromo - Estrategia

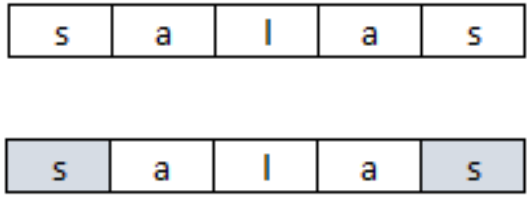
- La estrategia de resolución del problema por Divide y Conquista se puede describir de la siguiente manera:

Caso Base	Si la secuencia tiene longitud cero o uno, la solución trivial es devolver verdadero dado que la secuencia se considera palíndromo.
Subproblemas	Comparar los extremos de la secuencia, si no coinciden se devuelve falso, y sino se reduce la secuencia en dos extremos, y se resuelve si es palíndromo esa secuencia menor.
Combinación	Devolver el resultado de la secuencia de n-2.

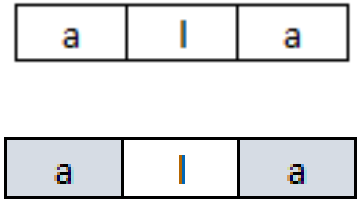
Determinar si una palabra Palíndromo - Estrategia

- A partir de la estrategia anterior, a partir de un ejemplo podemos validar la misma:

Ejemplo "*Verdadero*"



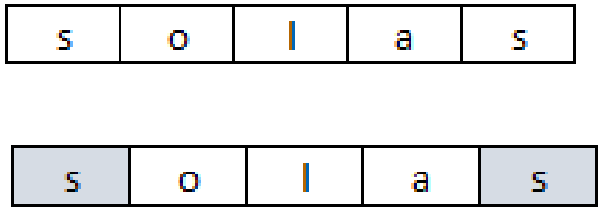
Suproblema 1:



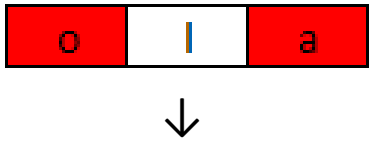
Suproblema 2 (sería el caso base) :

→ Devuelve "*Verdadero*"

Ejemplo "*Falso*"



Suproblema 1:



↓

Al comparar los extremos son distintos, entonces devuelve "*Falso*"

UADE

Determinar si una palabra Palíndromo - Algoritmo

ALGORITMO EsPalindromo

Entrada: **S**: secuencia o palabra sobre la cual determinar si es palíndromo o no; **inicio**, **fin**: índices a trabajar sobre la secuencia

Salida: **verdadero** o **falso** si la secuencia es Palíndromo o no

```
sí (inicio >= fin)
  devolver verdadero
 fin si
 sino
      sí S[inicio] <> S[fin]
        devolver falso
       fin si
       sino
          devolver EsPalindromo (S, inicio+1, fin-1)
       fin sino
    fin sino
fin EsPalindromo
```

Palabra Palíndromo – Complejidad Temporal

```
ALGORITMO EsPalindromo

Entrada: S: secuencia o palabra sobre la cual determinar si es
palíndromo o no; inicio, fin: índices a trabajar sobre la secuencia

Salida: verdadero o falso si la secuencia es Palíndromo o no

    sí (inicio >= fin)
      devolver verdadero
    fin si
    sino
        sí S[inicio] <> S[fin]
          devolver falso
        fin si
        sino
            devolver EsPalindromo (S, inicio+1, fin-1)
        fin sino
      fin sino
fin EsPalindromo
```

Es un caso de resta:

$$T(n) = \begin{cases} c & \text{si } 0 \leq n < b \\ aT(n-b) + p(n) & \text{si } n \geq b \end{cases}$$

a=1 ; b = 2; k=0

$$T(n) \in \begin{cases} \Theta(n^k) & \text{si } a < 1 \\ \Theta(n^{k+1}) & \text{si } a = 1 \\ \Theta(a^{n \; div \; b}) & \text{si } a > 1 \end{cases}$$

Como a = 1 entonces la complejidad temporal es: $\mathcal{O}(n)$

¡Muchas gracias!

UADE