Tema 5- Diseño Recursivo y Eficiente

Germán Moltó
Escuela Técnica Superior de Ingeniería Informática
Universidad Politécnica de Valencia

Tema 5- Diseño Recursivo y Eficiente

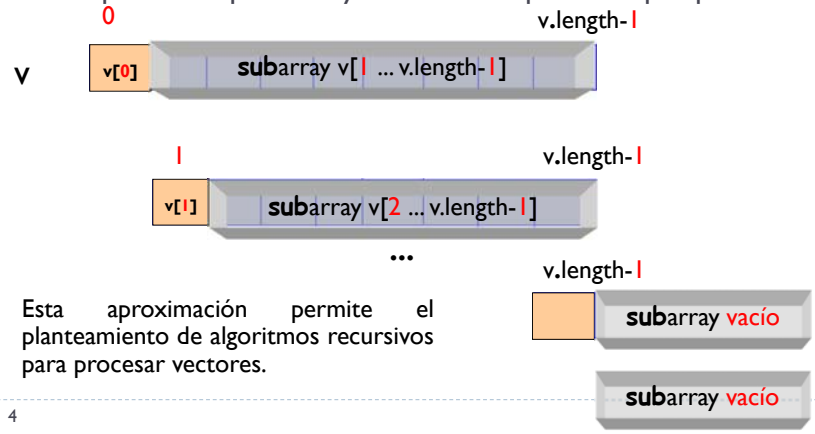
Índice general:

Etapas del diseño recursivo

1. Definición de la cabecera del método:
   - Perfil del método
   - Talla del problema
2. Análisis de Casos:
   - Hacer explícito los casos base y general de la recursión, estableciendo para cada uno de ellos las instrucciones que los resuelven.
3. Transcripción del Análisis de Casos (a Java)
4. Validación del diseño:
   - Comprobar que en su caso general, las llamadas que se realizan resuelven el mismo problema para tallas menores hasta alcanzar el caso base (Prueba de Terminación).

Descomposición recursiva ascendente de un vector

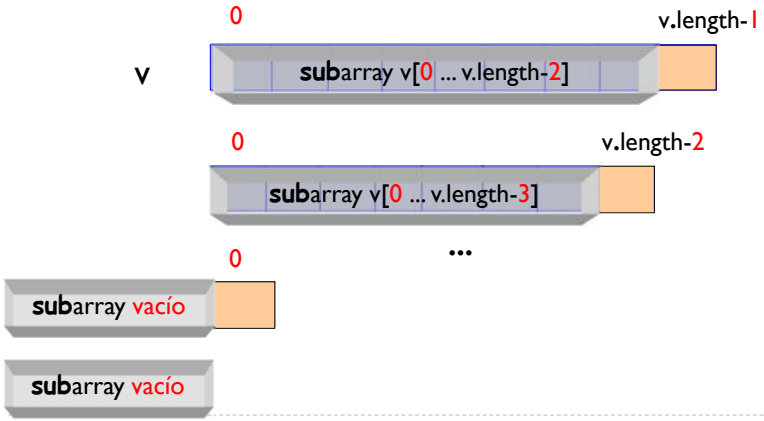
- Un vector se puede descomponer de mánera lógica en dos partes diferenciadas:
  - Componente a procesar y resto de componentes por procesar.



- Esta aproximación permite el planteamiento de algoritmos recursivos para procesar vectores.

Descomposición recursiva descendente de un vector

- Es posible aplicar el mismo patrón al recorrido descendente de un vector.



Diseño de un Método Recursivo de Recorrido de un Array

- Parámetros formales del método:
  1. El vector de objetos (T[] **v**) (nótese el uso de genericidad)
  2. La posición que marca el **inicio** del recorrido en cada llamada
  3. La posición que marca el **fin** del recorrido en cada llamada
- Talla de v[inicio ... fin]= fin – inicio +1
- Tipo de descomposición recursiva:
  - Ascendente: *inicio* se incrementa en cada llamada hasta que supera a *fin* en el caso base: inicio == fin + 1 (inicio>fin)
    $0 \leq inicio \leq v.length$ **AND** $fin == v.length - 1$
  - Descendente: *fin* se decrementa en cada llamada hasta que supera a *inicio* en el caso base: fin == inicio – 1 (fin<inicio)
    $-1 \leq fin \leq v.length-1$ **AND** $inicio == 0$

Ejemplo de Recorrido Recursivo Ascendente de un Vector

```
/** 0 <= inicio <= v.length AND fin == v.length-1 */
private static <T> void recorrer(T v[], int inicio, int fin) {
  if ( inicio <= fin ){
    procesar(v[inicio]);
    recorrer(v, inicio + 1, fin);
  }
}
public static <T> void recorrer(T v[]) {
  recorrer( v, 0, v.length - 1);
}
```


Si en cada llamada recursiva se le pasa el vector como argumento. ¿La pila de recursión se llenará enseguida para vectores de gran tamaño?

¿Qué tipo de recursión es?

- Se ha diseñado un método privado y un método **vía** o **lanzadera** público que permite facilitar su uso para que trabaje sobre todo un vector.

Ejercicio: Suma Recursiva de Vector

- Diseñar un método recursivo que calcule la suma de las componentes de un vector de enteros v:

```
// 0 ≤ inicio ≤ v.length AND fin == v.length-1
private static int sumarArray(int v[] , int  inicio, int fin) {
        …
}
// sumarArray(v, inicio, fin) == ∑i=inicio...fin v[i]
```

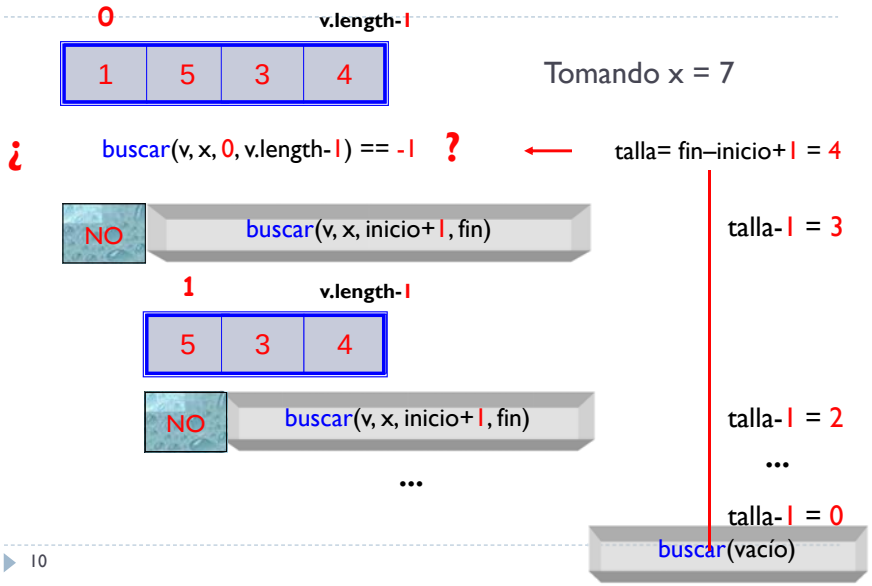
- Implementar el método general, el método vía y realizar un análisis del diseño recursivo.

Búsqueda Recursiva Secuencial: Descripción

- Diseñar un método recursivo que dados un objeto x y un array de objetos v de tipos compatibles, obtenga la posición de la primera aparición de x en v
- Si el elemento no pertenece al vector entonces se devolverá -1

```
// 0 <= inicio <= v.length AND fin = v.length -1
private static <T> int buscar(T v[] ,T x , int  inicio, int  fin){
        ...
}
```


Búsqueda Recursiva Secuencial: Traza Animada



Busqueda Recursiva Secuencial

```
private static <T> int buscar(T v[], T x, int inicio, int fin){
    if ( inicio>fin) return -1;
   else {
        if (v[inicio].equals(x) ) return inicio;
        else return buscar(v, x, inicio+1, fin);
   }
}
public static <T> int buscar(T v[], T x){
        return buscar(v, x, 0, v.length-1);
}
```

- ¿Qué tipo de recursión presenta la búsqueda recursiva secuencial? ¿Cuánto decrece la talla del problema en cada llamada recursiva? ¿Existen instancias significativas?

Búsqueda Binaria Recursiva: Descripción

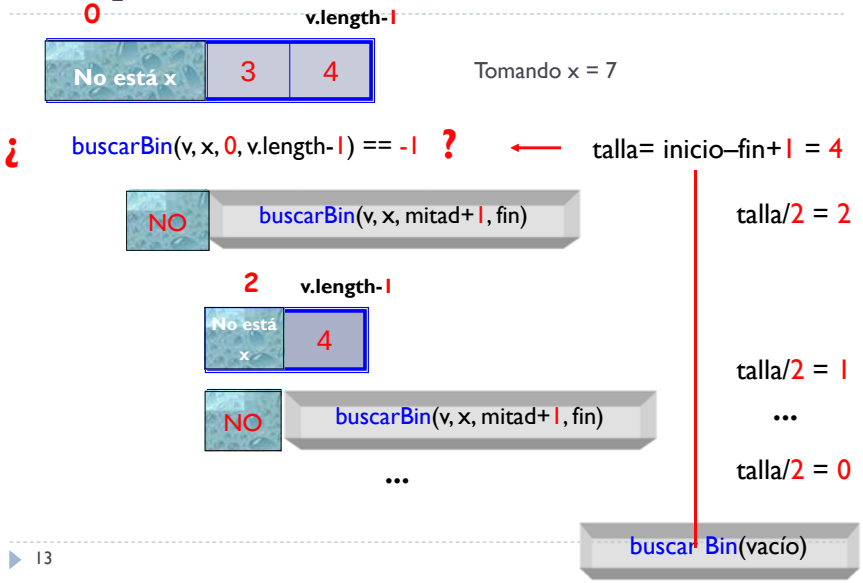
- Si el vector está ordenado, la búsqueda puede acelerarse.
- Busqueda Binaria Recursiva.
  - Aprovechar la ordenación para guiar el proceso de búsqueda
  - Evaluar el elemento central del vector y decidir por dónde debe continuar la búsqueda

¿Qué característica deberán cumplir los objetos del vector?

- La clase de los objetos implementará el interfaz **Comparable<E>** definiendo una relación de orden entre los objetos.

```
private static <T extends Comparable<T>>
        int  buscarBin(T v[] , T x , int  inicio, int  fin );
```

Búsqueda Binaria Recursiva: Traza



Búsqueda Binaria vs Búsqueda Secuencial

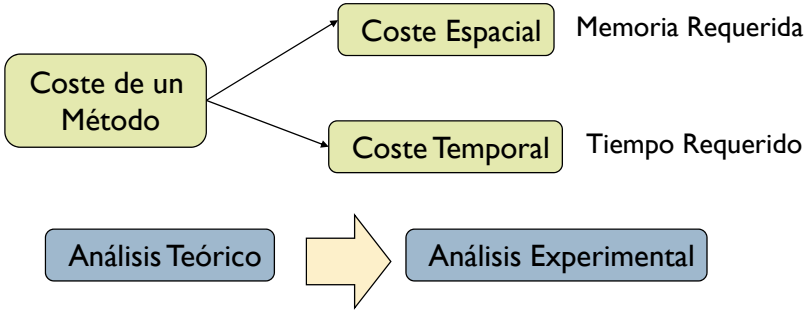
- Variación de la talla en el peor de los casos:
  - La **búsqueda binaria** consigue **reducir a la mitad** la talla del problema en cada llamada recursiva.
  - La **búsqueda secuencial** consigue **reducir en una unidad** la talla del problema en cada llamada recursiva
- La búsqueda binaria es mucho más rápida que la búsqueda secuencial.
  - Pero solo es aplicable cuando el vector está ordenado.
- Además, recuerda que únicamente se puede ordenar un vector cuando existe una relación de orden entre los elementos, es decir, que su clase implementa la interfaz **Comparable<E>**.

Búsqueda Binaria Recursiva: Implementación

```
private static <T extends Comparable<T>> int
   buscarBin(T v[] ,T x , int  inicio , int  fin) {
   if ( inicio > fin ) return - 1;
   else { int mitad = (inicio + fin) / 2;
    int cmp = v[mitad].compareTo(x) ;
    if ( cmp == 0 )  return mitad;
    else if ( cmp < 0 )  return buscarBin(v, x, mitad+1, fin);
          else    return buscarBin(v, x, inicio, mitad-1);
   }}
```

- La especificación del método requiere que el vector de sea de objetos cuya clase implemente Comparable<T>.

Coste de un Método

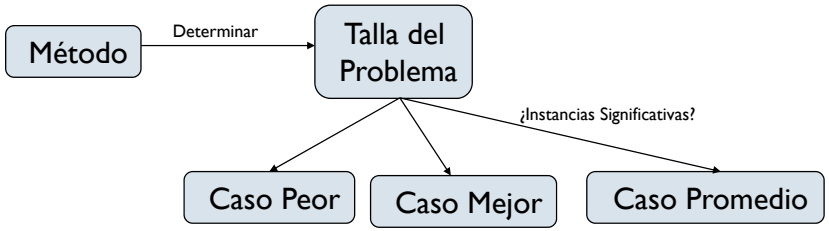


- El coste temporal de un método recursivo va ligado al tipo de recursión (taxonomía), lo que permite incorporar la eficiencia a la estrategia de diseño.

Complejidad Temporal: Recordatorio de PRG

- Detección de la **Instrucción Crítica** (asignación, comparación, etc.), aquella que se ejecuta por lo menos con tanta frecuencia como cualquier otra del método.
- Determinar la **talla del problema** que define la cantidad de datos a procesar.
- Determinar las **Instancia del Problema** (a partir de una talla fija):
  - Conjunto de configuraciones de los datos de entrada para las que el comportamiento del método (y su coste) es el mismo.
- Las instancias del problema dan lugar a la detección del **caso mejor** y del **caso peor** (fijando la talla del problema a un valor > 0).

Complejidad Temporal: Instancias Significativas



- **Recorrido** de un array:
  - No tiene instancias significativas. Se debe recorrer el vector.
- **Busqueda** en un array:
  - Sí tiene instancias significativas:
    - Caso Peor: El elemento no está en el array: $T^{P}_{buscar}(n)$
    - Caso Mejor: Elemento en la primera posición: $T^{M}_{buscar}(n)$
    - Caso Promedio: Comportamiento general del método

Complejidad Temporal: Notación Asintótica (I)

- Consideremos una función que trabaja sobre un vector v de n elementos. Para cada elemento realiza un conjunto de k operaciones básicas (asignación, aritmética, etc.).

```
public static <T> void procesaVector(T v[]){
   for (int i = 0 ; i<v.length; i++){ // k operaciones básicas }
}
```

Longitud del vector	1	2	3	…	n
Número de pasos	1*k	2*k	3*k	…	n*k

- El tiempo de ejecución del método crece de manera **lineal** (al mismo ritmo que) con el tamaño del vector.

Complejidad Temporal: Notación Asintótica (II)

- Consideremos una función que trabaja sobre un vector v de n elementos. Para cada elemento realiza un conjunto de k operaciones básicas.

```
public static <T> void procesaVector2(T v[]){
   for (int i = 0; i < v.length ; i++) {
         for (int j = 0 ; j<v.length; j++){ // k operaciones básicas }
   }}
```

Longitud del vector	1	2	3	…	n
Número de pasos	1*k	4*k	9*k	…	n^2*k

- El tiempo de ejecución crece de manera **cuadrática** con el tamaño del vector.

Complejidad Temporal: Notación Asintótica (III)

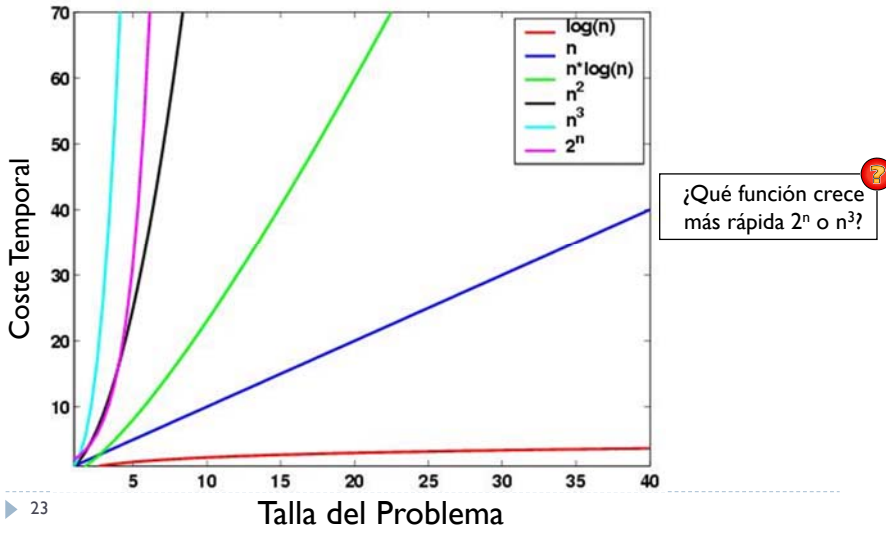
- Uso de expresiones de Complejidad Asintótica:
  $T_{metodo}(n) = 54*n^2 + 13*n$
  $T_{metodo}(n) \in \Theta(n^2)$
- Para valores suficientemente grandes de la talla, el valor de la función de complejidad esta completamente determinado por su término dominante.
- La notación asintótica permite expresar el coste asintótico de un método:
  - $O(f(n))$: Conjunto de funciones en n que son como máximo del orden de f(n):
  - $\Omega(f(n))$: Conjunto de funciones en n que son como mínimo del orden de f(n).
  - $\Theta(f(n))$: Conjunto de funciones en n que son exactamente del orden de f(n).

Notación Asintótica:

Nombre	Notación Asintótica
Constante	$\Theta(1)$
Logarítmica	$\Theta(\log_2 n)$
Lineal	$\Theta(n)$
n * log(n)	$\Theta(n*\log_2 n)$
Cuadrática	$\Theta(n^2)$
Cúbica	$\Theta(n^3)$
Exponencial	$\Theta(2^n)$

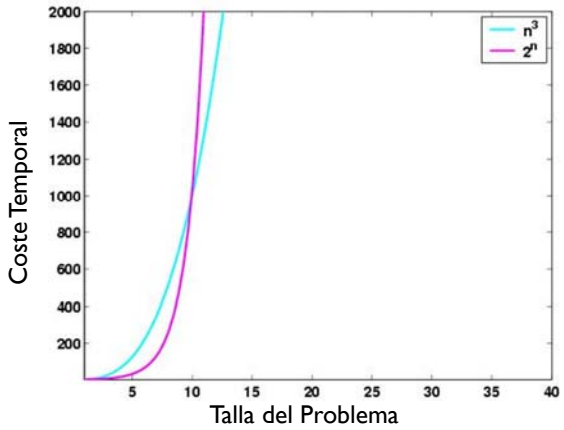
- El coste más interesante de todos es el **constante** puesto que es **independiente de la talla del problema**.
  - Aunque el tamaño del problema sea muy grande el algoritmo con coste constante no tardará mucho más (visión informal).

Representación Gráfica de Funciones



¿Qué función crece más rápida 2^n o n^3?

Cúbico vs Exponencial



- Para tamaños de problema suficientemente elevados es más costoso el exponencial que el cúbico.

Metodología General para el Análisis del Coste Temporal de un Método

1. Determinar la **talla del problema** y expresarla en función de una o más variables del método a analizar.
2. Determinar las posibles **instancias significativas** del problema para una talla concreta (Caso Mejor y Peor).
3. Determinar la **complejidad asintótica** para cada una de las instancias significativas del problema.
   1. Si no existen instancias significativas o coinciden sus cotas de complejidad: $T_{metodo}(talla) \in \Theta(f(talla))$
   2. Sino:
      - Si $T^{P}_{metodo}(talla) \in \Theta(f(talla))$ entonces $T_{metodo}(talla) \in O(f(talla))$
      - En el caso que $T^{M}_{metodo}(talla) \in \Theta(g(talla))$ entonces $T_{metodo}(talla) \in \Omega(g(talla))$

Función Complejidad de un Método Recursivo: Aplicación al Factorial

```
static int factorial(int n){
        if (n == 0) return 1;
        else return n*factorial(n-1);
}
```

- **Complejidad Espacial**:
  - Talla del Problema: n
  - Complejidad Asintótica: Θ(n)
    - Número de Registros de Activación (R.A) proporcional al valor de n.
  - Un análisis experimental revelará el tamaño del R.A.
- **Complejidad Temporal**: No hay instancias significativas.
  - Ecuaciones de Recurrencia:
    - $T_{factorial}(n = 0) = k$
    - $T_{factorial}(n > 0) = 1 * T_{factorial}(n - 1) + k'$
  - La resolución de las Ecuaciones de Recurrencia dan lugar a la función de la Complejidad Temporal que deberá ser acotada por el Coste Asintótico.

Interpretando las Funciones de Recurrencia (I)

$T_{metodo}(casoBase(talla)) = k$
$T_{metodo}(casoGeneral(talla)) = a * T_{metodo}(n\ op\ c) + b$

- Siempre hay dos ecuaciones de recurrencia (caso base y caso general de la recursión).
  - Si además hay instancias significativas entonces tenemos 4 ecuaciones de recurrencia (2 para el mejor caso y 2 para el caso peor). Las del mejor caso suelen ser triviales.
- Las ecuaciones de recurrencia se expresan en función de la talla del problema.
- k: Expresa el coste de realizar las acciones cuando se alcanza el caso base (si depende de la talla del problema se expresa como k*talla).

Interpretando las Funciones de Recurrencia (II)

$T_{metodo}(casoBase(talla)) = k$
$T_{metodo}(casoGeneral(talla)) = a * T_{factorial}(n\ op\ c) + b$

- a: Indica el número de llamadas recursivas que se realizan.
- op: Expresa cómo decrece la talla del problema:
  - Progresión aritmética (restando en al menos una unidad)
  - Progresión geométrica (dividiendo la talla del problema)
- b: Indica la sobrecarga, es decir, el coste de las acciones realizadas aparte de la llamada recursiva.
  - Comprobar si la sobrecarga depende de la talla del problema (expresada como *b*talla*) o no (expresada como *k*).
- Para acotar las ecuaciones se utilizan teoremas.

Teoremas de Resolución de Ecuaciones de Recurrencia (I)

- **Teorema 1:**
  - Talla decrece en progresión aritmética y sobrecarga independiente de la talla

  Sea $T_f(x) = a*T_f(x - c) + b$, donde b>=1, entonces:
  - Si a = 1, $T_f(x) \in \Theta(x)$
  - Si a > 1, $T_f(x) \in \Theta(a^{x/c})$
- **Teorema 2:**
  - Talla decrece en progresión aritmética y sobrecarga dependiente de la talla

  Sea $T_f(x) = a*T_f(x - c) + b*x + d$, donde b>=1 y d>=1:
  - Si a = 1, $T_f(x) \in \Theta(x^2)$
  - Si a > 1, $T_f(x) \in \Theta(a^{x/c})$

Teoremas de Resolución de Ecuaciones de Recurrencia (II)

- **Teorema 3:**
  - Talla decrece en progresión geométrica y sobrecarga independiente de la talla

  Sea $T_f(x) = a*T_f(x/c) + b$, entonces:
  - Si a = 1, $T_f(x) \in \Theta(\log_c(x))$
  - Si a > 1, $T_f(x) \in \Theta(x^{\log_c(a)})$
- **Teorema 4:**
  - Talla decrece en progresión geométrica y sobrecarga dependiente de la talla

  Sea $T_f(x) = a*T_f(x/c) + b*x + d$, entonces:
  - Si a < c, $T_f(x) \in \Theta(x)$
  - Si a = c, $T_f(x) \in \Theta(x*\log_c(x))$
  - Si a > c, $T_f(x) \in \Theta(x^{\log_c(a)})$

Ejemplos de Aplicación de Teoremas:

- Ecuación de Recurrencia de Factorial:
  - $T_{factorial}(n = 0) = k'$
  - $T_{factorial}(n > 0) = 1 * T_{factorial}(n - 1) + k$
  - Resolvemos y acotamos aplicando el Teorema 1 con a = c = 1

    $T_{factorial}(n) \in \Theta(n)$
- Ecuación de Recurrencia de Hanoi (No Visto en Teoría):
  - $T_{hanoi}(n = 1) = T_{moverDisco} = k'$
  - $T_{hanoi}(n > 1) = 2 * T_{hanoi}(n - 1) + k$
  - Resolvemos y acotamos aplicando el Teorema 1 con a = 2 y c=1

    $T_{Hanoi}(n) \in \Theta(2^n)$

Cálculo de la Complejidad Temporal de la Búsqueda Secuencial Recursiva

- Algoritmo de Búsqueda Secuencial Recursiva

```
private static <T> int buscar(T v[] ,T x , int  inicio , int  fin)
{
  if ( inicio > fin )  return -1;
  else {
        if (v[inicio].equals(x)) return inicio;
        else return buscar(v, x, inicio + 1, fin);
 }
}
```

- Asumimos invocación a **buscar**(v, x, 0, v.length-1);

Cálculo de la Complejidad Temporal de la Búsqueda Secuencial Recursiva (II)

1. Talla del Problema (en función de los argumentos).
   talla = fin – inicio + 1 (n° de elementos del vector)
2. Instancias Significativas
   - Caso Mejor:
     - El elemento buscado está en la primera posición del vector
   - Caso Peor:
     - El elemento buscado NO está en el vector:
3. Ecuaciones de Recurrencia:
   - Para el caso mejor:
     - $T^M_{buscar}(talla) = k$
   - Para el caso peor:
     - $T^P_{buscar}(talla = 0) = k$
     - $T^P_{buscar}(talla > 0) = 1*T^P_{buscar}(talla-1) + k'$

Cálculo de la Complejidad Temporal de la Búsqueda Secuencial Recursiva (III)

4. Complejidad Asintótica para cada Instancia Significativa (Utilizando el Teorema 1 con a = c = 1):
   - $T^P_{buscar}(talla) \in \Theta(talla)$
   - $T^M_{buscar}(talla) \in \Theta(1)$

   $T_{buscar}(talla) \in O(talla)$
   $T_{buscar}(talla) \in \Omega(1)$

   - La ecuación de recurrencia para el caso mejor es tan sencilla que no precisa de teorema para su resolución.

La Estrategia de Reducción Logarítmica

- La estrategia de diseño **óptima** de un método **recursivo lineal** con sobrecarga constante consiste en **decrementar la talla** del problema **geométricamente**.
- **Teorema 1:**

Sea $T_f(x) = a*T_f(x - c) + b$, donde b>=1, entonces:

- Si a = 1, $T_f(x) \in \Theta(x)$

- **Teorema 3:**

Sea $T_f(x) = a*T_f(x/c) + b$, entonces:

- Si a = 1, $T_f(x) \in \Theta(\log_c(x))$

- Es más ventajoso el coste logarítmico que el coste lineal ya que aumenta de manera más lenta con respecto al tamaño del problema.

Reducción Logarítmica: La Búsqueda Binaria (I)

```
private static <T extends Comparable<T>> int buscarBin(T
  v[] ,T x , int  inicio , int  fin) {
  if ( inicio > fin ) return - 1;
  else { int mitad = (inicio + fin) / 2;
   int cmp = v[mitad].compareTo(x) ;
   if ( cmp == 0 )  return mitad;
   else if ( cmp < 0 )  return buscarBin(v, x, mitad+1, fin);
        else    return buscarBin(v, x, inicio, mitad-1);
  }}
```

Complejidad Temporal de la Búsqueda Binaria (I)

1. Talla del Problema.
   talla = fin – inicio + 1
2. Instancias Significativas.
   - Caso Mejor: El elemento buscado está justo en la posición central del vector
   - Caso Peor: El elemento buscado NO está en el vector
3. Ecuaciones de Recurrencia
   - Caso Mejor
     - $T^M_{buscarBin}(talla) = k$
   - Caso Peor:
     - $T^P_{buscarBin}(talla = 0) = k$
     - $T^P_{buscarBin}(talla > 0) = 1*T^P_{buscarBin}(talla/2) + k$

Complejidad Temporal de la Búsqueda Binaria (II)

4. Complejidad Asintótica para cada Instancia Significativa (Usando el Teorema 3 con a = 1, c= 2):
   - $T^M_{buscarBin}(talla) \in \Theta(1)$
   - $T^P_{buscarBin}(talla) \in \Theta(\log_2(talla))$

   $T_{buscarBin}(talla) \in O(\log_2(talla))$
   $T_{buscarBin}(talla) \in \Omega(1)$

- La ecuación de recurrencia para el caso mejor es tan sencilla que no precisa de teorema para su resolución.

Otros Ejemplos de Reducciones Logarítmicas

1. Búsqueda en un vector ordenado de tamaño n:
   - Búsqueda recursiva secuencial: $\Theta(n)$.
   - Búsqueda binaria: $\Theta(\log_2(n))$.
2. Multiplicación de dos números naturales a y b:

```
/ ** a >= 0 AND b >= 0 */
static int multiplicar(int a, int b) {
    if ( a == 0 ) return 0;
    else return multiplicar(a - 1, b) + b;
}
```

Realizar una traza de llamadas para multiplicar(4,5)
¿En qué orden se producen las llamadas recursivas?
¿En qué orden finalizan?

   - Ecuaciones de Recurrencia: (Talla: magnitud de a).
     - $T_{multiplicar}(a = 0) = k'$
     - $T_{multiplicar}(a > 0) = 1 * T_{multiplicar}(a - 1) + k$
   - Coste Temporal (Teorema 1 con a = 1 y c = 1):
     - $T_{multiplicar}(A) \in \Theta(A)$

Otros Ejemplos de Reducciones Logarítmicas (II)

2. Multiplicación de dos números naturales a y b:

```
static int multiplicarRL(int a, int b) {
    if ( a == 0 )              return 0;
    else {
          int resLlamada = multiplicarRL(a/2, b);
          if ( a % 2 == 0 ) return resLlamada * 2;
          else                        return resLlamada * 2 + b;
}}
```

   - Ecuaciones de Recurrencia:
     - $T_{multiplicarRL}(a = 0) = k'$
     - $T_{multiplicarRL}(a > 0) = 1 * T_{multiplicarRL}(a/2) + k$
   - Coste Temporal (Teorema 3 con a = 1 y c = 2):
     - $T_{multiplicarRL}(A) \in \Theta(\log_2(A))$

Recursión Múltiple vs Recursión Lineal

- Si un método presenta Recursión Múltiple, se debe exigir que los subproblemas representados por las llamadas recursivas sean **disjuntos** (ej. Fibonacci).
- En general la recursión múltiple implica un coste más elevado que la recursión lineal (teoremas).
- Casos especiales:
  - Ante sobrecarga constante (b), si la recursión es múltiple con tantas llamadas recursivas como subproblemas a resolver (a = c), se obtiene una misma cota **(lineal)** que el problema resuelto mediante reducción aritmética de la talla y recursión lineal. (Teoremas 1 y 3).
  - Ante sobrecarga lineal con la talla y recursión lineal, se puede obtener una cota temporal peor **(cuadrática)** que con recursión múltiple y división geométrica de la talla **(n*log(n))**. (Teoremas 2 y 4).