Implementación recursiva de la búsqueda binaria

Algoritmo que decide si un valor está o no en un vector. El vector ordenado en orden estrictamente creciente. (simplificación respecto a la versión de los apuntes)

Especificación.

$\{0 \leq num \leq longitud(v) \wedge ord(v, num)\}$

fun buscaBin (TElem v[], int num, TElem x) return bool b

$\{b \equiv \exists i : 0 \leq i < num : v[i] = x\}$

donde : $ord(v, num) \equiv \forall i : 0 \leq i < num - 1 : v[i] < v[i+1]$

```
typedef int TElem;

bool buscaBin( TElem v[], int num, TElem x ) {
   bool esta;
// cuerpo de la función
   return esta;
}
```

Implementación recursiva de la búsqueda binaria

Planteamiento recursivo.

- Primera aproximación: Comparar el último elemento del vector con el elemento buscado y si no es igual buscar en el vector $v[0..num-1]$.
- Descomposición de la postcondición:
  $\exists i : 0 \leq i < num : v[i] = x \equiv$
  $\exists i : 0 \leq i < num - 1 : v[i] = x \vee v[num-1] = x$
- Estrategia recursiva:
  buscaBin(v,num,x) = buscaBin(v,num-1,x) || v[num -1] == x
- El coste de esta solución será lineal.
- ¿Podemos aprovechar el hecho de que el vector está ordenado para mejorar el coste del algoritmo?.

Implementación recursiva de la búsqueda binaria

Planteamiento recursivo.

- Segunda aproximación: Dividir el vector por la mitad y buscar sólo en una de las dos mitades.
- Descomposición de la postcondición:
  $\exists i : 0 \leq i < num : v[i] = x \equiv$
  $\quad \exists i : 0 \leq i \leq num/2 : v[i] = x \vee$
  $\quad \exists i : num/2 + 1 \leq i < num : v[i] = x$
- Estrategia recursiva:
  buscaBin(v,num,x) =
  $\quad$ buscaBin(v1,num/2,x) || buscaBin(v2,num/2,x)
- Problema: esta estrategia exige copiar el vector para realizar la llamada. No es aceptable.
- Podriamos evitar realizar la copia del vector si pudiésemos referirnos a una parte cualquiera del vector original.

Implementación recursiva de la búsqueda binaria

Planteamiento recursivo.

- Realizamos una generalización de la función original que utiliza dos parámetros que se refieren al principio y al final del vector.
- Especificación de la función auxiliar:
  $\{0 \leq p \leq long(v) \wedge -1 \leq f < long(v) \wedge p \leq f + 1$
  $\wedge ord(v, p, f)\}$
  fun buscaBin (TElem v[], int p, int f, TElem x) return bool b
  $\{b \equiv \exists i : p \leq i \leq f : v[i] = x\}$

```
typedef int TElem;

bool buscaBin( TElem v[], int p, int f, TElem x ) {
  bool esta;

// cuerpo de la función

  return esta;
}
```

Implementación recursiva de la búsqueda binaria

Planteamiento recursivo.

- Tercera aproximación: Realizar la recursión sobre la función generalizada. Dividir el vector por la mitad y buscar solo en una de las dos partes.
- Descomposición de la postcondición:
  $\exists i : p \leq i \leq f : v[i] = x \equiv$
  $\quad \exists i : p \leq i \leq (p+f)/2 : v[i] = x \vee$
  $\quad \exists i : (p+f)/2+1 \leq i \leq f : v[i] = x$
- Evitamos buscar en una de las dos partes:
  $v[(p+f)/2] \geq x \wedge \exists i : p \leq i \leq (p+f)/2 : v[i] = x \vee$
  $v[(p+f)/2] < x \wedge \exists i : (p+f)/2+1 \leq i \leq f : v[i] = x$
- Estrategia recursiva:
  $buscaBin(v, p, f, x) =$

$$\begin{cases} v[(p+f)/2] \geq x & buscaBin(v, p, (p+f)/2, x) \\ v[(p+f)/2] < x & buscaBin(v, (p+f)/2+1, f, x) \end{cases}$$

Implementación recursiva de la búsqueda binaria

Problema. La longitud del subvector a considerar (función cota) no disminuye si el vector tiene un único elemento:

- Caso $v[(p+f)/2] \geq x$:
- Función sucesor: $s_1(v, p, f, x) = (v, p, (p+f)/2, x)$
- Longitud del subvector que estamos considerando: $f - p + 1$
- Longitud del subvector a considerar en la llamada recursiva: $(p+f)/2 - p + 1$
- Si $p == f$ (vector de un único elemento) el subvector de la llamada recursiva es el mismo que el de la llamada principal.

Solución. Considerar como caso base el vector de un elemento.

Implementación recursiva de la búsqueda binaria

Análisis de casos.

- Solución directa:
  - $p > f$ el resultado es *false*.
  - $p == f$ el resultado es $v[p] == v[f]$
- Distinción de casos:
  1. $d_1(v, p, f, x) : p > f$ y $d_2(v, p, f, x) = p == f$
  2. $\neg d(v, p, f, x) : p < f$
- Se cubren todos los casos:

$$p > f \vee p == f \vee p < f \equiv true$$

Solución en el caso directo.

$$A_1 \equiv b = false;$$
$$A_2 \equiv b = v[p] == v[q]$$

Implementación recursiva de la búsqueda binaria

Descomposición recursiva ($p < f$).

$$\begin{array}{ll} v[(p+f)/2] \geq x: & s_1(v,p,f,x) = (v,p,(p+f)/2,x) \\ v[(p+f)/2] < x: & s_2(v,p,f,x) = (v,(p+f)/2+1,f,x) \end{array}$$

Función de acotación y terminación.

$$t(v,p,f,x) = f - p + 1$$

Se cumple que:

- $v[(p+f)/2] \geq x : (p+f)/2 - p + 1 < f - p + 1 \equiv (p+f)/2 < f$

$$\begin{cases} par((p+f)/2) & (p+f)/2 < f \equiv (p+f) < 2f \equiv p < f \\ impar((p+f)/2) & (p+f)/2 < f \equiv p + f - 1 < 2f \equiv p - 1 < f \end{cases}$$

- $v[(p+f)/2] < x : f - (p+f)/2 + 1 < f - p + 1$ Semejante a la anterior.

Implementación recursiva de la búsqueda binaria

LLamada recursiva.

$$\begin{cases} v[(p+f)/2] \geq x: & 0 \leq p \leq long(v) \wedge -1 \leq (p+f)/2 \leq long(v) \\ & \wedge p \leq (p+f)/2+1 \\ v[(p+f)/2] < x: & 0 \leq (p+f)/2+1 \leq long(v) \wedge \\ & -1 \leq f \leq long(v) \wedge (p+f)/2+1 \leq f+1 \end{cases}$$

Función de combinación. No hay, se trata de una función recursiva final.

Escritura del caso recursivo.

```
if (v[(p+f)/2] >= x) esta = buscaBin(v,p,(p+f)/2,x);
else esta = buscaBin(v,(p+f)/2+1,f,x);
```

La función auxiliar obtenida es:

```
bool buscaBin ( int v[], int p, int f, int x ) {
  bool esta;
  if ( p > f ) esta = false;
     else if (p == f) esta = (v[p] == v[f]);
  else  {
     if (v[(p + f) / 2] >= x)
                esta = buscaBin(v,p,(p + f) / 2,x);
     else esta = buscaBin(v,(p + f) / 2+1,f,x);

  return esta;
}
```

La función principal se reduce a llamar a la función auxiliar con los parámetros adecuados.

```
bool buscaBin ( int v[], int num, int x ) {
  return buscaBin(v,0,num-1,x);
}
```

Mejoras que se pueden introducir en la función:

- La función puede terminar cuando se encuentra el valor, sin necesidad de seguir buscando. Consideramos como caso base cuando se encuentra el valor y eliminamos este elemento de las llamadas recursivas.
- Optimizamos el cálculo de la componente del medio del vector.
- La variable *esta* no es necesaria.

```
bool buscaBin ( int v[], int p, int f, int x ) {
  if ( p > f ) return false;
  else if (p == f) return  (v[p] == v[f]);
  else  {
     int m = (p + f) / 2;
     if (v[m] == x) return true;
     else if (v[m] > x)  return buscaBin(v,p,m-1,x);
     else return buscaBin(v,m+1,f,x);
  return esta;
}
```

Análisis de la complejidad de algoritmos recursivos

- Para analizar la complejidad de un algoritmo recursivo debemos analizar la complejidad de lo que se hace en una llamada recursiva, y estimar el número de llamadas recursivas que se realizarán.
- Introducimos el concepto de *ecuaciones de recurrencia*
- Cálculo del factorial.

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

- Ecuaciones de recurrencia:

$$T(n) = \begin{cases} c_0 & \text{si } n = 0 \\ c_1 + T(n-1) & \text{si } n > 0 \end{cases}$$

- Multiplicación por el método del campesino egipcio.

```
int prod ( int a, int b )
{
  int r;
  if      ( b == 0 )                    r = 0;
  else if ( b == 1 )                    r = a;
  else if ( b > 1 && (b %2 == 0) )
            r = prod(2*a, b/2);
  else if ( b > 1 && (b %2 == 1) )
           r = prod(2*a, b/2) + a;
  return r;
}
```

- Ecuaciones de recurrencia: ($n = b$ tamaño del problema)

$$T(n) = \begin{cases} c_0 & \text{si } n = 0, 1 \\ c_1 + T(n/2) & \text{si } n > 1 \end{cases}$$

- Números de fibonacci.

```
int fib( int n )
{
  int r;

  if      ( n == 0 ) r = 0;
  else if ( n == 1 ) r = 1;
  else if ( n >  1 ) r = fib(n-1) + fib(n-2);

  return r;
}
```

- Ecuaciones de recurrencia:

$$T(n) = \begin{cases} c_0 & \text{si } n = 0, 1 \\ T(n-1) + T(n-2) + c_1, & \text{si } n > 1 \end{cases}$$

Despliegue de recurrencias

Método para obtener una *fórmula explícita* del orden de complejidad de una recurrencia.

1. **Despliegue**. Sustituimos las apariciones de T en la recurrencia tantas veces como sea necesario hasta encontrar una fórmula que dependa del número de llamadas recursivas k.
2. **Postulado**. Obtenemos el valor de k que nos permite alcanzar un caso directo y, en la fórmula paramétrica, se sustituimos k por ese valor y la referencia recursiva T por la complejidad del caso directo. La fórmula obtenida es la expresión explícita del orden de complejidad.
3. **Demostración**. La fórmula explícita así obtenida sólo es correcta si la recurrencia para el caso recursivo también es válida para el caso directo. Se comprueba demostrando por inducción que la fórmula obtenida cumple las ecuaciones de recurrencia.

- Factorial
  - Ecuaciones

$$T(n) = \begin{cases} c_0 & \text{si } n = 0 \\ c_1 + T(n-1) & \text{si } n > 0 \end{cases}$$

  - Despliegue

$$\begin{aligned} T(n) &= c_1 + T(n-1) \\ &= c_1 + c_1 + T(n-2) \\ &= c_1 + c_1 + c_1 + T(n-3) \\ &\ldots \\ &= c_1 * k + T(n-k) \end{aligned}$$

  - Postulado
    El caso directo se tiene para $n = 0$

$$n - k = 0 \Leftrightarrow k = n$$

$$T(n) = c_1 n + T(n-n) = c_1 n + T(0) = c_1 n + c_0$$

    Por lo tanto $T(n) \in O(n)$

Resultados teóricos

Disminución del tamaño del problema por sustracción.

- Cuando:
  1. la descomposición recursiva se obtiene restando una cierta cantidad constante
  2. el caso directo tiene coste constante
  3. la preparación de las llamadas y de combinación de los resultados tiene coste polinómico, entonces la ecuación de recurrencias será de la forma:

$$T(n) = \begin{cases} c_0 & \text{si } 0 \leq n < n_0 \\ a * T(n-b) + c * n^k & \text{si } n \geq n_0 \end{cases}$$

- Se puede demostrar (libro Diseño de programas. Formalismo y Abstracción. R. Peña. Pag: 17):

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

- Vemos que, cuando el tamaño del problema disminuye por sustracción,
  - En recursión simple ($a = 1$) el coste es polinómico y viene dado por el producto del coste de cada llamada ($c * n^k$) y el coste lineal de la recursión (n).
  - En recursión múltiple ($a > 1$), por muy grande que sea b, el coste siempre es exponencial.

Disminución del tamaño del problema por división.

- Cuando:
  1. la descomposición recursiva se obtiene dividiendo por una cierta cantidad constante
  2. el caso directo tiene coste constante
  3. la preparación de las llamadas y de combinación de los resultados tiene coste polinómico, entonces la ecuación de recurrencias será de la forma:

$$T(n) = \begin{cases} cn^k & \text{si } 1 \leq n < b \\ a * T(n/b) + c * n^k & \text{si } n \geq b \end{cases}$$

- Se puede demostrar (libro Diseño de programas. Formalismo y Abstracción. R. Peña. Pag: 20):

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

- Si $a \leq b^k$ la complejidad depende de n^k que es el término que proviene de $c * n^k$ en la ecuación de recurrencias, y, por lo tanto, la complejidad de un algoritmo de este tipo se puede mejorar disminuyendo la complejidad de la preparación de las llamadas y la combinación de los resultados.
- Si $a > b^k$ las mejoras en la eficiencia se pueden conseguir
  - disminuyendo el número de llamadas recursivas a o aumentando el factor de disminución del tamaño de los datos b, o bien
  - optimizando la preparación de las llamadas y combinación de los resultados, pues, si esto hace disminuir k suficientemente, podemos pasar a uno de los otros casos: $a = b^k$ o incluso $a < b^k$.

Ejemplos

- Suma recursiva de un vector de enteros.

```
//fun sumaVec (int v[], int num) return int r
int sumaVec ( int v[], int num ) {
// Pre: longitud(v) ≥ num
  int  r;

  if ( num <= 0 )
     r = 0;
  else if ( num >  0 )
     r = sumaVec(v, num-1) + v[num-1];

  return r;
// Post: r = ∑ i : 0 ≤ i < num : v[i]
}
```

- Recurrencias (tamaño de los datos $n = num$):

$$T(n) = \begin{cases} c_1 & \text{si } n = 0 \\ T(n-1) + c & \text{si } n > 0 \end{cases}$$

Ejemplos

- Búsqueda binaria.

```
bool buscaBin ( int v[], int p, int f, int x ) {
  bool esta;
  if ( p > f )
     esta = false;
  else  {
     int m = (p + f) / 2;
     if (v[m] >= x)  esta = buscaBin(v,p,m,x);
     else esta = buscaBin(v,m+1,f,x);

  return esta;
}
```

- Recurrencias (Tamaño de los datos: $n = num$):

$$T(n) = \begin{cases} c_1 & \text{si } n = 0, 1 \\ T(n/2) + c & \text{si } n > 1 \end{cases}$$