Ejercicios Resueltos del tema 10

Contenido

Cálculo del valor acumulado de los elementos de una secuencia

Partimos de una secuencia y queremos acumular los valores que la forman mediante un determinado operador binario. Queremos encontrar formulaciones recursivas de algoritmos que resuelvan el problema. Ejemplos concretos de este problema general son:

1. Sumar de los elementos de una lista
2. Factorial de un número entero n
3. Máximo de los n elementos de un array de reales dt que contiene n elementos.
4. Suma de los dígitos del entero a en base r
5. Número de dígitos del entero a en base r
6. Buscar si el elemento e está contenido en lista ls.
    a. Asumir que la lista no está ordenada
    b. Asumir que la lista está ordenada
7. Encontrar el índice del elemento e en lista ls y si no lo contiene entonces devolver -1.
    a. Asumir que la lista no está ordenada
    b. Asumir que la lista está ordenada
8. Buscar si existe un e contenido en lista ls que cumpla el predicado $p(x)$
9. Calcular a^n
10. Obtener la representación del entero e en base r
11. Buscar si todos los dígitos del entero e que cumplen el predicado $p(x)$

Todos los enunciados concretos anteriores comparten los mismos elementos y ello permite diseñar algoritmos recursivos que pueden ser ajustados a cada problema particular.

El primer paso es ver que todos ellos toman como entrada una secuencia. Una secuencia s de tamaño n es una sucesión de elementos y lo escribimos como:

$$s = (e_0, e_1, \dots, e_{n-1})$$

Hay n elementos siendo e_0 el primero y e_{n-1} el último. Si en a la secuencia anterior le podemos asociar el operador $s[i]$ tal que dado $i: 0..n-1$ nos devuelve el elemento e_i entonces decimos que la secuencia es indexable. En cualquier caso dado una secuencia, indexable o no, podemos descomponerla en las formas siguientes:

1. $s = e + s1$: Esquema sufijo
2. $s = s1 + e$: Esquema prefijo
3. $s = s1 + s2$: Esquema subsecuencias.
4. $s = e1 + s1 + e2$: Esquema subsecuencia central

Dónde $s, s1, s2$ son secuencias y $e, e1$ elementos de la misma. En cada problema particular habrá que aportar los operadores necesarios para calcular $s1, s2, e, e1$ a partir de s e indicar cuando se puede aplicar o no. Los esquemas anteriores nos dan las formas para buscar definiciones recursivas en problemas que tomen secuencias como entrada.

Si la secuencia es indexable y representamos por $s[i]$ el elemento en posición $i: 0..n-1$ y $s[i,j]$ la subsecuencia que incluye los elementos de la posición i hasta la j incluyendo la i pero no la j con $i, j: 0..n-1 | j \geq i$. En las secuencias indexables los elementos $s1, s2, e, e1$ se pueden calcular fácilmente. La forma de conseguirlo es generalizar la secuencia la secuencia indexable ls de manera que podamos usar los cuatro esquemas anteriores:

1. (ls, i): Representa $ls[i, n]$. Generalización sufija. Aquí $e = ls[i], s1 = (ls, i+1)$
2. (ls, i): Representa $ls[0, i]$. Generalización prefija. Aquí $e = ls[i-1]$, $s1 = (ls, i-1)$
3. (ls, i, j): Representa $ls[i, j]$. Generalización subsecuencias. Aquí $k = \frac{i+j}{2}$, $s1 = (ls, j, k)$, $s2 = (ls, k, j)$
4. (ls, i, j): Representa $ls[i, j]$. Generalización secuencia central. Aquí $e1 = ls[i]$, $s1 = (ls, i+1, j-1)$, $e2 = ls[j-1]$

Con los cuatro esquemas anteriores podemos abordar los problemas que toman una secuencia como punto de partida. El primer paso es identificar la secuencia y decidir si es fácilmente indexable. Si no es indexable buscar uno de los esquemas de descomposición anteriores y si lo es una de las generalizaciones.

Veamos para problema de los anteriores cual es la secuencia, si es indexable o no y el esquema correspondiente:

1. La lista es una secuencia indexable con $n = ls.size()$, $ls[i] = ls.get(i)$. Resolvamos el problema por los cuatro esquemas.
2. Sabiendo que $n! = n(n-1)(n-2) \dots 1$ la secuencia es evidente. Podemos asumirla como no indexable (aunque podría ser indexable también) y escogemos el esquema de descomposición sufijo sabiendo que $n! = n(n-1)!$.
3. Un array de n elementos que podemos representar por (a, n) es una secuencia indexable con $n, a[i]$ ya definidos. Resolvamos el problema por los cuatro esquemas.
4. Un entero a en base r puede ser escrito como $a = d_k d_{k-1} \dots d_0 | r$. Siendo d_i los dígitos correspondientes. La secuencia es evidente con $n = k + 1$. La podemos tomar como no indexable y escoger le esquema prefijo sabiendo que $a = \frac{a}{r} + a\%r$.
5. Observando que $a^n = aaa \dots a$ la secuencia es evidente. Podemos usar cualquiera de los esquemas y considerar la secuencia como indexada o no. Veremos las soluciones abajo. Veremos otras secuencias asociadas a este problema.

Para cada problema veamos los diferentes esquemas y el código que va resultando. Sólo haremos algunos dejando el resto como ejercicio.

Secuencias Indexables. Generalización sufija.

Suma

$$suma(ls) = sm(ls, 0)$$
$$sm(ls, i) = \begin{cases} 0, & i = n \\ sm(ls, i+1) + ls[i], & i < n \end{cases}$$

Producto

$$producto(ls) = pd(ls, 0)$$
$$pd(ls, i) = \begin{cases} 1, & i = n \\ pd(ls, i+1) * ls[i], & i < n \end{cases}$$

Máximo

$$max(ls) = \begin{cases} \perp, & n = 0 \\ mx(ls, 0), & n > 0 \end{cases}$$
$$mx(ls, i) = \begin{cases} ls[i], & i = n-1 \\ m2(mx(ls, i+1), ls[i]), & i < n-1 \end{cases}$$
$$m2(a, b) = a > b ? a : b$$

Mínimo

$$min(ls) = \begin{cases} \perp, & n = 0 \\ mn(ls, 0), & n > 0 \end{cases}$$

$$mn(ls,i) = \begin{cases} ls[i], & i = n-1 \\ m3(mn(ls,i+1), ls[i]), & i < n-1 \end{cases}$$
$$m3(a,b) = a < b ?\, a : b$$

Contiene

$$contiene(ls,e) = ct(ls,e,0)$$
$$ct(ls,e,i) = \begin{cases} false, & i = n \\ ls[i] = e \,||\, ct(ls,e,i+1), & i < n \end{cases}$$

Discutir en este esquema las implicaciones de las propiedades de cortocircuito del operador || y por lo tanto las ventajas del orden de la llamada recursiva.

Existe

$$existe(ls,e) = ex(ls,p,0)$$
$$ct(ls,p,i) = \begin{cases} false, & i = n \\ p(ls[i]) \,||\, ct(ls,p,i+1), & i < n \end{cases}$$

Discutir en este esquema las implicaciones de las propiedades de cortocircuito del operador || y por lo tanto las ventajas del orden de la llamada recursiva.

Posición

$$index(ls,e) = in(ls,e,0)$$
$$in(ls,e,i) = \begin{cases} -1, & i = n \\ i, & i < n, ls[i] = e \\ in(ls,e,i+1), & i < n, ls[i] \neq e \end{cases}$$

Secuencias Indexables. Generalización prefija.

Suma

$$suma(ls) = sm(ls,n)$$
$$sm(ls,i) = \begin{cases} 0, & i = 0 \\ sm(ls,i-1) + ls[i-1], & i > 0 \end{cases}$$

Producto

$$producto(ls) = pd(ls,n)$$
$$pd(ls,i) = \begin{cases} 1, & i = 0 \\ pd(ls,i-11) * ls[i-1], & i > 0 \end{cases}$$

Máximo

$$max(ls) = \begin{cases} \perp, & n = 0 \\ mx(ls, 0), & n > 0 \end{cases}$$
$$mx(ls, i) = \begin{cases} ls[i-1], & i = 1 \\ m2(mx(ls, i-1), ls[i-1]), & i > 1 \end{cases}$$
$$m2(a, b) = a > b ? a : b$$

Mínimo

$$min(ls) = \begin{cases} \perp, & n = 0 \\ mn(ls, 0), & n > 0 \end{cases}$$
$$mn(ls, i) = \begin{cases} ls[i-1], & i = 1 \\ m3(mn(ls, i-1), ls[i-1]), & i > 1 \end{cases}$$
$$m3(a, b) = a < b ? a : b$$

Contiene

$$contiene(ls, e) = ct(ls, e, n)$$
$$ct(ls, e, i) = \begin{cases} false, & i = 0 \\ ls[i-1] = e || ct(ls, e, i-1), & i > 0 \end{cases}$$

Discutir en este esquema las implicaciones de las propiedades de cortocircuito del operador || y por lo tanto las ventajas del orden de la llamada recursiva.

Posición

$$index(ls, e) = in(ls, e, n)$$
$$in(ls, e, i) = \begin{cases} -1, & i = 0 \\ i-1, & i > 0, ls[i-1] = e \\ in(ls, e, i-1), & i > 0, ls[i-1] \neq e \end{cases}$$

Existe

$$existe(ls, p) = ex(ls, p, n)$$
$$ex(ls, p, i) = \begin{cases} false, & i = 0 \\ p(ls[i-1]) || ct(ls, p, i-1), & i > 0 \end{cases}$$

Discutir en este esquema las implicaciones de las propiedades de cortocircuito del operador || y por lo tanto las ventajas del orden de la llamada recursiva.

Secuencias Indexables. Generalización en subsecuencias

Suma

$$suma(ls) = sm(ls, 0, n)$$

$$sm(ls, i, j) = \begin{cases} 0, & j - i = 0 \\ ls[i], & j - i = 1 \\ sm(ls, i, k) + sm(ls, k, j), & j - i > 1 \end{cases}$$

$$k = (i + j)/2$$

Discutir porqué hacen falta los casos base $j - i = 0, j - i = 1$.

Producto

$$producto(ls) = pd(ls, 0, n)$$

$$pd(ls, i, j) = \begin{cases} 1, & j - i = 0 \\ ls[i], & j - i = 1 \\ pd(ls, i, k) * pd(ls, k, j), & j - i > 1 \end{cases}$$

$$k = (i + j)/2$$

Máximo

$$max(ls) = \begin{cases} \perp, & n = 0 \\ mx(ls, 0, n), & n > 0 \end{cases}$$

$$mx(ls, i, j) = \begin{cases} ls[i], & j - i = 1 \\ m2(ls[i], ls[i + 1]), & j - i = 2 \\ m2(mx(ls, i, k), mx(ls, k, j), & j - i > 2 \end{cases}$$

$$m2(a, b) = a > b ? a : b$$

$$k = (i + j)/2$$

Discutir porqué hacen falta ambos casos base

Mínimo

$$min(ls) = \begin{cases} \perp, & n = 0 \\ mn(ls, 0), & n > 0 \end{cases}$$

$$mn(ls, i) = \begin{cases} ls[i], & j - i = 1 \\ m3(ls[i], ls[i + 1]), & j - i = 2 \\ m3(mn(ls, i, k), mn(ls, k, j)), & j - i > 2 \end{cases}$$

$$m3(a, b) = a < b ? a : b$$

$$k = (i + j)/2$$

Contiene. Secuencia no ordenada

$$contiene(ls, e) = ct(ls, e, 0, n)$$

$$ct(ls, e, i) = \begin{cases} false, & j - i = 0 \\ true, & j - i = 1, ls[i] = e \\ false, & j - i = 1, ls[i] \neq e \\ ct(ls, e, i, k) || ct(ls, e, k, j), & j - i > 1 \end{cases}$$
$$k = (i + j)/2$$

Contiene. Secuencia ordenada. Búsqueda binaria

$$contiene(ls, e) = ct(ls, e, 0, n)$$

$$ct(ls, e, i, j) = \begin{cases} false, & j - i = 0 \\ false, & j - i = 1, ls[i] \neq e \\ true, & j - i = 1, ls[i] = e \\ true, & j - i > 1, e = ls[k] \\ ct(ls, e, k + 1, j), & j - i > 1, e > ls[k] \\ ct(ls, e, i, k), & j - i > 1, e < ls[k] \end{cases}$$
$$k = (i + j)/2$$

Discutir en este esquema la posibilidad de eliminar algún caso base.

Posición. Secuencia no ordenada

$$index(ls, e) = in(ls, e, 0, n)$$

$$in(ls, e, i, j) = \begin{cases} -1, & j - i = 0 \\ i, & j - i = 1, ls[i] = e \\ -1, & j - i = 1, ls[i] \neq e \\ \max\left(in(ls, e, i, k), in(ls, e, k, j)\right), & j - i > 1 \end{cases}$$
$$k = (i + j)/2$$

Discutir en este esquema las implicaciones del diseño de la función $max(a, b)$ para este problema particular.

Posición. Secuencia ordenada

$$index(ls, e) = in(ls, e, 0, n)$$

$$in(ls,e,i,j) = \begin{cases} -1, & j-i=0 \\ i, & j-i=1, ls[i]=e \\ -1, & j-i=1, ls[i]\neq e \\ k, & j-i>1, ls[k]=e \\ in(ls,e,k+1,j), & j-i>1, e>ls[k] \\ in(ls,e,i,k)), & j-i>1, e<ls[k] \end{cases}$$
$$k=(i+j)/2$$

Existe.

$$existe(ls,p) = ex(ls,p,0,n)$$
$$ex(ls,p,i,j) = \begin{cases} false, & j-i=0 \\ true, & j-i=1, p(ls[i]) \\ false, & j-i=1, !\,p(ls[i]) \\ ct(ls,p,i,k)||ct(ls,p,k,j), & j-i>1 \end{cases}$$
$$k=(i+j)/2$$

Secuencias Indexables. Generalización en subsecuencia central

Se deja como ejercicio

Secuencias no indexables. Generalización prefijo

Factorial de n

$$n! = \begin{cases} 1, & n=0 \\ (n-1)! * n, & n>0 \end{cases}$$

Suma de los dígitos de un entero e en base r

$$sd(e) = \begin{cases} 0, & e=0 \\ sd(e/r)+e\%r, & e>0 \end{cases}$$

Número de dígitos de un entero e en base r

$$nd(e) = \begin{cases} 0, & e=0 \\ nd(e/r)+1, & e>0 \end{cases}$$

Representación de un entero e en base r

$$rp(e) = \begin{cases} "", & e = 0 \\ rp(e/r) +_L "e\%r", & e > 0 \end{cases}$$

Potencia a^n

$$a^n = \begin{cases} 1, & n = 0 \\ a^{n-1} * a, & n > 0 \end{cases}$$

Todos los dígitos de un entero e en base r cumplen el predicado p(d)

$$alld(e,p) = \begin{cases} true, & e = 0 \\ p(e\%r) \wedge alld(e/r), & e > 0 \end{cases}$$

Secuencias no indexables. Generalización en subsecuencias

Potencia a^n

$$a^n = \begin{cases} 1, & n = 0 \\ (a^{n/2})^2, & n > 0, n\%2 = 0 \\ (a^{n/2})^2 * a, & n > 0, n\%2 \neq 0 \end{cases}$$

Secuencias. Esquemas recursivos finales

Dada una secuencia caracterizada por $sq = (x_o, s(x), d(x))$, dónde $d(x)$ es un predicado que especifica el dominio de la misma, $p(x)$ un predicado que indica los elementos que queremos acumular, $g(x)$ un valor calculado a partir de x que queremos acumular y $\bigotimes_I$ un operador asociativo por la izquierda con elemento neutro e.

Con los elementos anteriores un esquema recursivo final para acumular los elementos de la secuencia con ese operador es:

$$f(x) = fg(x, e)$$

$$fg(x,a) = \begin{cases} a\,, & !\,d(x) \\ fg(s(x), a \bigotimes_I g(x)), & d(x) \wedge p(x) \\ fg(s(x), a), & d(x) \wedge !\,p(x) \end{cases}$$

Y el correspondiente esquema iterativo es:

```
A f(T x){
      A a = e;
      while(d(x)){
            if(p(x)) a = a ⊗I g(x);
            x = s(x);
      }
```

```
        return a;
}
```

Para usar los esquemas anteriores tenemos que definir la secuencia mediante su dominio. Es decir proporcionando un predicado que abarca a todos los elementos de la secuencia y sólo a ellos.

Hemos de tener en cuenta que cualquier algoritmo con una sola llamada recursiva define una secuencia que podemos utilizar para encontrar esquemas recursivos finales y sus equivalentes iterativos.

Suma de los elementos de una lista

En las listas la secuencia considerada es $(0,1,2 \dots, n-1)$, por lo que $i_0 = 0, s(i) = i + 1$. Aquí el operador $\bigotimes_I$ es la suma y los valores a acumular $ls[i]$.

$$suma(ls) = sm(ls, 0, 0)$$
$$sm(ls, i, a) = \begin{cases} a, & i = n \\ sm(ls, i+1, a + ls[i]), & i < n \end{cases}$$

Producto de los elementos de una lista

$$producto(ls) = pd(ls, 0, 1)$$
$$pd(ls, i, a) = \begin{cases} a, & i = n \\ pd(ls, i+1, a * ls[i]), & i < n \end{cases}$$

Máximo de los elementos de una lista

$$max(ls) = \begin{cases} \perp, & n = 0 \\ mx(ls, 0, ls[0]), & n > 0 \end{cases}$$
$$mx(ls, i, a) = \begin{cases} a, & i = n \\ mx(ls, i+1, a), & i < n, a \geq ls[i] \\ mx(ls, i+1, ls[i]), & i < n, a < ls[i] \end{cases}$$

Mínimo de los elementos de una lista

$$min(ls) = \begin{cases} \perp, & n = 0 \\ mn(ls, 0, ls[0]), & n > 0 \end{cases}$$
$$mn(ls, i, a) = \begin{cases} a, & i = n \\ mn(ls, i+1, a), & i < n, a \leq ls[i] \\ mn(ls, i+1, ls[i]), & i < n, a > ls[i] \end{cases}$$

Contiene. Secuencia no ordenada

$$contiene(ls, e) = ct(ls, e, 0)$$

$$ct(ls, e, i) = \begin{cases} false, & i = n \\ true, & i < n, ls[i] = e \\ ct(ls, e, i+1), & i < n, ls[i] \neq e \end{cases}$$

Discutir porqué no es necesario un acumulador

Contiene. Secuencia ordenada

$$contiene(ls, e) = ct(ls, e, 0, n)$$

$$ct(ls, e, i, j) = \begin{cases} false, & j - i = 0 \\ ls[i] = e, & j - i = 1 \\ true, & j - i > 1, e = ls[k] \\ ct(ls, e, i, k), & j - i > 1, e < ls[k] \\ ct(ls, e, k+1, j), & j - i > 1, e > ls[k] \end{cases}$$

$$k = (i + j)/2$$

Discutir porqué no es necesario un acumulador. Como ejemplo el correspondiente esquema iterativo es:

```
boolean contiene(List<T> ls, T e, Comparator<T> cmp){
      boolean r = false;
      int i = 0
      int j = ls.size();
      int k;
      while(j-i>0){
            if(j-i==1){
                  r = cmp.compare(e,ls.get(i))==0;
                  break;
            }
            k = (i+j)/2;
            if(cmp.compare(e, ls.get(k))==0){
                  r = true;
                  break;
            }
            if(cmp.compare(e, ls.get(k))<0){
                  j = k;
            }
            if(cmp.compare(e, ls.get(k))>0){
                  i = k+1;
            }
      }
      return r;
}
```

Hemos tenido que concretar el orden en el *comparator* y escoger tipo de secuencia, en este caso una lista (alternativamente podríamos tener un array, un *String*, etc.) y según el lenguaje final los detalles serán diferentes.

Posición. Secuencia no ordenada

$$index(ls, e) = in(ls, e, 0, -1)$$
$$in(ls, e, i) = \begin{cases} -1, & i = n \\ i, & i < n, ls[i] = e \\ n(ls, e, i+1), & i < n, ls[i] \neq e \end{cases}$$

Discutir porqué no es necesario un acumulador

Posición. Secuencia ordenada

$$index(ls, e, i, j) = \begin{cases} -1, & j - i = 0 \\ i, & j - i = 1, ls[i] = e \\ -1, & j - i = 1, ls[i] \neq e \\ k, & j - i > 1, ls[k] = e \\ in(ls, e, k+1, j), & j - i > 1, e > ls[k] \\ in(ls, e, i, k), & j - i > 1, e < ls[k] \end{cases}$$
$$k = (i + j)/2$$

Discutir porqué no es necesario un acumulador

Existe

$$existe(ls, p) = ex(ls, p, 0)$$
$$ex(ls, p, i) = \begin{cases} false, & i = n \\ true, & i < n, p(ls[i]) \\ ct(ls, p, i+1), & i < n, !\,p(ls[i]) \end{cases}$$

Discutir porqué no es necesario un acumulador

Factorial de n

$$n! = fac(n, 1)$$
$$fac(n, a) = \begin{cases} a, & n = 0 \\ fac(n-1, a * n), & n > 0 \end{cases}$$

Suma de los dígitos de un entero e en base r

En los enteros una secuencia posible es $(e, e/r, e/r^2 \ldots, 1)$, por lo que $s(e) = e/r$ y el dominio $d(e) > 0$. Aquí el operador binario es la suma y los valores a acumular $e\%r$.

$$sd(e) = sdg(e, 0)$$
$$sdg(e, a) = \begin{cases} a, & e = 0 \\ sdg\left(\frac{e}{r}, a + e\%r\right), & e > 0 \end{cases}$$

Número de dígitos de un entero e en base r

Para poder considerar el dominio $d(e) > 0$ tenemos que considerar el caso particular $e = 0$.

$$nd(e) = \begin{cases} 1, & e = 0 \\ ndg(e, 0), & e > 0 \end{cases}$$
$$ndg(e, a) = \begin{cases} a, & e = 0 \\ ndg\left(\frac{e}{r}, a + 1\right), & e > 0 \end{cases}$$

Representación de un entero e en base r

El operador $+_L$ es asociativo por la izquierda pero no conmutativo por lo que hay que colocar los operandos adecuadamente.

$$rp(e) = \begin{cases} "0", & e = 0 \\ rpg(e, ""), & e > 0 \end{cases}$$
$$rpg(e, a) = \begin{cases} a, & e = 0 \\ rpg(\frac{e}{r}, "e\%r" +_L a), & e > 0 \end{cases}$$

Potencia b^n. Considerando la secuencia bbbbbbb

La secuencia considerada es $s(n) = n - 1$ y el dominio $d(n) > 0$. Es decir $(n, n - 1, \dots, 1)$. Aquí el operador binario es el producto y los valores a acumular n.

$$b^n = pot(b, n, 1)$$
$$pot(b, n, a) = \begin{cases} a, & n = 0 \\ pot(b, n - 1, b * a), & n > 0 \end{cases}$$

Potencia a^n.

Consideremos la secuencia definida por $s(n, y) = \left(\frac{n}{2}, y^2\right)$. Es decir $((n, y), \left(\frac{n}{2}, y^2\right), \left(\frac{n}{4}, y^4\right), \dots)$ con dominio $d(n, y) = n > 0$. El filtro dado por el predicado $p(n, y) = n\%2 \neq 2$ y acumular los valores de y con el operador producto cuyo elemento neutro es 1.

La secuencia se deduce de las propiedades:

$$n = \sum_{i=0}^{p} d_i 2^i$$

$$a^n = a^{\sum_{i=0}^{p} d_i 2^i} = \prod_{i=0}^{p} a^{d_i 2^i}$$

$$d_0 = \begin{cases} 0, & n\%2 = 0 \\ 1, & n\%2 = 1 \end{cases}$$

$$a^{2^{i+1}} = a^{2^i 2} = (a^{2^i})^2$$

El esquema es:

$$pot(a,n) = pt(a,n,a,1)$$

$$pt(a,n,y,u) = \begin{cases} u, & n = 0 \\ pt(a,\frac{n}{2},y^2,u), & n > 0, n\%2 = 0 \\ pt(a,\frac{n}{2},y^2,u*y), & n > 0, n\%2 \neq 0 \end{cases}$$

Todos los dígitos de un entero e en base r cumplen el predicado p(d)

$$alld(e,p) = \begin{cases} p(0), & e = 0 \\ ad\left(\frac{e}{r},p\right), & e > 0 \end{cases}$$

$$ad(e,p) = \begin{cases} true, & e = 0 \\ false, & e > 0, !\,p(e\%r) \\ ad\left(\frac{e}{r},p\right), & e > 0, r, p(e\%r) \end{cases}$$

Discutir porqué no es necesario un acumulador

Esquemas sobre tipos recursivos

Dada la definición recursiva de un árbol de la forma:

$$t = \begin{cases} empty() \\ unary(e) \\ binary(e,t1,t2) \end{cases}$$

Junto con la definición anterior es conveniente definir predicados como $\mathrm{isEmpty(t)}$, $\mathrm{isUnary(t)}$, $\mathrm{isBinary(t)}$ que, respectivamente, deciden si t es un árbol vacío, unario o binario. Adicionalmente si un árbol t es binario asumimos que ha sido construido de la forma $\mathrm{t} = \mathrm{binary(e, t1, t2)}$ y por lo tanto nos podremos referir a sus elementos $\mathrm{e, t1, t2}$. Igualmente para los caso unario.
Diseñar algoritmos recursivos para calcular los siguientes objetivos:

1. Número de elementos del árbol
2. Suma de los elementos del árbol que son pares (asumiendo que los elementos son enteros)
3. Obtener una lista con todos los elementos del árbol. Discuta las posibles formas de hacerlo.
4. Calcular la altura de un árbol (la altura es el número de pasos desde la raíz a la hoja más lejana y por lo tanto la altura de un árbol con un solo elemento es 0)
5. Contiene al elemento *a*
6. Profundidad de un vértice en un árbol o -1 si el vértice no está contenido (la profundidad es el número de pasos desde la raíz al vértice dado)
7. Comprobar si dos árboles son iguales

Número de elementos

$$ne(t) = \begin{cases} 0, & isEmpty(t) \\ 1, & isUnary(t) \\ ne(t1) + ne(t2) + 1, & isBinary(t) \end{cases}$$

Suma de los elementos del árbol que son pares (asumiendo que los elementos son enteros)

$$sum(t) = \begin{cases} 0, & isEmpty(t) \\ 0, & isUnary(t), e\%2! = 0 \\ e, & isUnary(t), e\%2 == 0 \\ sum(t1) + sum(t2), & isBinary(t), e\%2! = 0 \\ sum(t1) + sum(t2) + e, & isBinary(t), e\%2 == 0 \end{cases}$$

Obtener una lista con todos los elementos del árbol. Discuta las posibles formas de hacerlo.

$$toList1(t) = \begin{cases} [], & isEmpty(t) \\ [e], & isUnary(t) \\ [e] + ne(t1) + ne(t2), & isBinary(t) \end{cases}$$

$$toList2(t) = \begin{cases} [], & isEmpty(t) \\ [e], & isUnary(t) \\ ne(t1) + [e] + ne(t2), & isBinary(t) \end{cases}$$

$$toList3(t) = \begin{cases} [], & isEmpty(t) \\ [e], & isUnary(t) \\ ne(t1) + ne(t2) + [e], & isBinary(t) \end{cases}$$

Discutir las diferencias entre las tres formas propuestas

Calcular la altura de un árbol (la altura es el número de pasos desde la raíz a la hoja más lejana y por lo tanto la altura de un árbol con un solo elemento es 0)

$$h(t) = \begin{cases} 0, & isEmpty(t) \\ 0, & isUnary(t) \\ max\ (h(t1), h(t2)) + 1, & isBinary(t) \end{cases}$$

Contiene al elemento a

$$ct(t, a) = \begin{cases} false, & isEmpty(t) \\ true, & isUnary(t), a = e \\ false, & isUnary(t), a \neq e \\ true, & isBinary(t), a = e \\ ct(t1, a)||ct(t2, a), & isBinary(t), a \neq e \end{cases}$$

Profundidad de un vértice a en un árbol o -1 si el vértice no está contenido en el árbol (la profundidad es el número de pasos desde la raíz al vértice dado)

$$pf(t, a) = pg(t, a, 0)$$

$$pg(t, a, r) = \begin{cases} -1, & isEmpty(t) \\ r, & isUnary(t), a = e \\ -1, & isUnary(t), a \neq e \\ r, & isBinary(t), a = e \\ \max\ (pg(t1, a, r + 1), pg(t2, a, r + 1)), & isBinary(t), a \neq e \end{cases}$$

Comprobar si dos árboles son iguales

$$eq(tu, tv) = \begin{cases} true, & isEmpty(tu), isEmpty(tu) \\ true, & isUnary(tu), isUnary(tv), eu = ev \\ false, & isUnary(tu), isUnary(tv), eu \neq ev \\ false, & isUnary(tu), isBinary(tv) \\ false, & isBinary(tu), isUnary(tv) \\ eu = ev, eq(tu1, tv1), eq(tu2, tv2), & isBinary(tu), isBinary(tv) \end{cases}$$

Dónde la coma representa el operador &&, $eu, tu1, tu2$ los elementos del árbol tu y de forma similar para el árbol tv.

La implementación en Java o C se deja como ejercicio.

Algoritmos a partir de definiciones recursivas

Encontrar algoritmos recursivos a partir de las definiciones recursivas siguientes:

1. $ab = \begin{cases} a + a(b-1), & b > 0 \\ 0, & b = 0 \end{cases}$
2. $xy = \begin{cases} 0, & y = 0 \\ x, & y = 1 \\ 2x\left(\frac{y}{2}\right), & y > 1, y\ es\ par \\ 2x\left(\frac{y}{2}\right) + x, & y > 1, y\ es\ impar \end{cases}$
3. $\begin{cases} \binom{n}{0} = \binom{n}{n} = 1, & n \geq 0 \\ \binom{n}{1} = \binom{n}{n-1} = n, & n \geq 1 \\ \binom{n}{k} = \binom{n}{n-k}, & n \geq k \\ \binom{n}{k} + \binom{n}{k+1} = \binom{n+1}{k+1}, & n > k \end{cases}$

1. *Versiones recursivas no final y final de 1*

   a. $m1(a,b) = \begin{cases} 0, & b = 0 \\ a + m1(a, b-1), & b > 0 \end{cases}$

   b. $m1(a,b) = m1f(a,b,0)$

   $m1f(a,b,u) = \begin{cases} a, & b = 0 \\ m1f(a, b-1, u+a), & b > 0 \end{cases}$

2. *Versiones recursivas no final y final de 2*

   a. $m2(x,y) = \begin{cases} 0, & y = 0 \\ x, & y = 1 \\ 2\ m2\left(x, \frac{y}{2}\right), & y > 1, y\ par \\ 2\ m2\left(x, \frac{y}{2}\right) + x, & y > 1, y\ impar \end{cases}$

   b. $m2(x,y) = m2g(x,y,x,0)$

$$m2g(x,y,p,a) = \begin{cases} a, & y = 0 \\ m2g\left(x, \frac{y}{2}, 2p, a\right), & y > 1, y\ par \\ m2\left(x, \frac{y}{2}, 2p, a+p\right), & y > 1, y\ impar \end{cases}$$

En esta última versión se ha buscado una nueva secuencia. El algoritmo no final define la secuencia $(y, \frac{y}{2}, \frac{y}{4}, \dots, 1)$ y a partir de ella hace el cálculo del producto que se puede expresar de la forma:

$$xy = x(\sum_{i=0}^{k} d_i 2^i) = \sum_{i=0}^{k} x d_i 2^i)$$

Dónde los d_i son los dígitos de y en su representación binaria. La expresión anterior podemos verla como la suma de los términos de la secuencia $(xd_i2^i, xd_i2^i, \dots, xd_k2^k)$ dónde aquellos con $d_i = 0$ pueden ser eliminados. La secuencia puede ser definida por los pares (y,p) con un primer elemento (y,x), elemento siguiente $s(y,p) = (\frac{y}{2}, 2p)$ y dominio $d(y,p) \equiv y > 0$.

La versión iterativa es:

```
long m2(long x, long y){
      long a = 0;
      long p = x;
      while(y>0){
            if(y%2!=0){
                a = a + p;
            }
            y = y/2;
            p = p+p;
      }
      return a;
}
```

3. Versiones recursivas no final de 3

$$nc(n,k) = \begin{cases} 1, & k = 0 || k = n \\ n, & k = 1 || k = n-1 \\ nc(n-1, \min(k, n-k)) + nc(n-1, \min(k-1, n-(k-1))), & n > 1 \end{cases}$$

$$dom(n,k) \equiv n \geq 0, n \geq k$$

Este algoritmo recursivo no es lineal por lo cual no podemos buscar una secuencia ampliada que al acumular sus valores nos proporcione una versión final.

La vía para obtener una versión iterativa (o su equivalente recursiva final) es usando una técnica *bottom-up* (como haremos en los ejemplos de recurrencias). Se trata de calcular progresivamente los valores de $nc(n,k)$ desde $(0,0)$ hasta (n,k) incrementando los valores de i,j en un orden que al calcular $nc(i,j)$ se hayan calculado previamente $n(i-1,j)$ y $n(i-1,j-1)$. Guardando los valores en una aplicación (Map) m con la operaciones $m(n,k)$ y $m+(n,k)$ una versión iterativa es:

```
long nc(long n, long k){
    long i = 0;
    long j = 0;
    Map m = {};
    while(i<=n && j<=k){
         if(j==i||j==0){
              m(i,j)=1;
         } else if(j==1||j==i-1){
              m(i,j)=i;
         } else {
              m(i,j)= m(i-1,j)+ m(i-1,j-1);
         }
         j=j+1;
         if(j>i){
              j=0;
              i=i+1;
         }
    }
    return m(n,k);
}
```

En la versión anterior se han guardado todos los valores calculados pero podemos buscar un algoritmo similar que guarde sólo los datos necesarios. Como vemos para calcular $m(i,j)$ sólo hacen falta los datos intermedios entre este valor y $m(i-1,j-1)$ en el orden especificado. El resto de los datos guardados no es necesario por lo que pueden ser eliminados o diseñar otra forma de guardar sólo los necesarios. Se deja como ejercicio concretar esos cambios.

Recurrencias

Dada la recurrencia $f_n = 2f_{n-1} - f_{n-3}, f_2 = 1, f_1 = 1, f_0 = 2$:

a. Diseñar un algoritmo recursivo, con y sin memoria.

b. Adaptar el problema al de la potencia entera.
c. Diseñar un algoritmo iterativo que calcule los valores de la recurrencia mediante el cálculo de abajo arriba de los valores de la recurrencia.
d. Diseñar un algoritmo iterativo a partir de la adaptación del problema al de la potencia entera

Resolvemos el primero y dejamos el segundo como ejercicio.

$$f(n) = \begin{cases} 2, & n = 0 \\ 1, & n = 1 \\ 1, & n = 2 \\ 2f(n-1) - f(n-3), & n > 2 \end{cases}$$

La implementación con memoria se deduce del esquema anterior. La adaptación al problema de la potencia es de la forma:

$$\begin{pmatrix} f(n+3) \\ f(n+2) \\ f(n+1) \end{pmatrix} = \begin{pmatrix} 2 & 0 & -1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} f(n+2) \\ f(n+1) \\ f(n) \end{pmatrix}$$

$$A = \begin{pmatrix} 2 & 0 & -1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}, v0 = \begin{pmatrix} 1 \\ 1 \\ 2 \end{pmatrix}$$

$$\begin{pmatrix} f(n+2) \\ f(n+1) \\ f(n) \end{pmatrix} = v = A^n v0, \qquad n \geq 0$$

Con los elementos anteriores el esquema del programa iterativo, adaptado del algoritmo para la potencia, resulta ser:

```
long rec1(int n){
     Matrix y, u;
     Vector v;
     y = A;
     u = I;
     v = V0;
     while( n > 0){
          if(n%2==1){
               u = u * y;
          }
          y = y * y;
          n = n/2;
     }
     v = u*V0;
     return v(0);
}
```

Para implementar el algoritmo necesitamos diseñar e implementar el tipo *Matrix* con las operaciones producto y cuadrado y el tipo *Vector,* una matriz con una sola columna. Hemos representado por `I` la matriz identidad y siendo `A, V0` los vistos arriba.

La versión iterativa obtenida a partir de la definición recursiva anterior podemos obtenerla con le invariante:

$$f(i+2) = a$$
$$f(i+1) = b$$
$$f(i) = c$$

Por lo que, si pretendemos aumentar la i de uno en uno:

$$\begin{pmatrix} a' \\ b' \\ c' \\ i' \end{pmatrix} = \begin{pmatrix} f(i'+2) \\ f(i'+1) \\ f(i') \\ i' \end{pmatrix} = \begin{pmatrix} f(i+3) \\ f(i+2) \\ f(i+1) \\ i+1 \end{pmatrix} = \begin{pmatrix} 2f(i+2) - f(i) \\ a \\ b \\ i+1 \end{pmatrix} = \begin{pmatrix} 2a-c \\ a \\ b \\ i+1 \end{pmatrix}$$

Es decir:

$$(a', b', c', i') \doteq (2a - c, a, b, i+1)$$

```
long rec2(int n){
     long a,b,c,i;
    (a,b,c,i)≐ (1,1,2,0);
     while( i <n){
          (a,b,c,i)≐ (2a-c,a,b,i+1);
     }
     return c;
}
```

La asignación paralela interior al bucle se puede implementar como:

```
a1 = 2a-c;
b1 = a;
c1 = b;
i1 = i+1;
a = a1;
b = b1;
c = c1;
i = i1;
```

Que se puede simplificar a:

```
b1 = a;
a = 2a-c;
c = b;
b = b1;
i = i+1;
```

La transformación anterior puede comprobarse obteniendo la asignación paralela equivalente.

El esquema resultante es:

```
long rec2(int n){
     long a,b,c,i,b1;
     a = 1;
     b = 1,
     c = 0;
     i = 0;
     while( i <n){
          b1 = a;
          a = 2a-c;
          c = b;
          b = b1;
          i = i+1;
     }
     return c;
}
```

Dada la recurrencia $f_n = 4f_{n-1} + f_{n-2} + f_{n-3}, f_2 = 1, f_1 = 1, f_0 = 2$:

a. Diseñar un algoritmo recursivo, con y sin memoria.
b. Adaptar el problema la de la potencia entera.
c. Diseñar un algoritmo iterativo que calcule los valores de la recurrencia mediante el cálculo de abajo arriba de los valores de la recurrencia.
d. Diseñar un algoritmo iterativo a partir de la adaptación del problema al de la potencia entera

Se deja como ejercicio

Diseño Iterativo

Diseñar algoritmos iterativos para resolver los siguientes problemas:

1. Factorial de un entero

2. Suma de los elementos de una lista de reales
3. Mezcal de dos lista ordenadas en otra también ordenada

El diseño iterativo parte de un Invariante sobre las variables de un problema generalizado. Sean x las variables del problema original e y las adicionales del problema generalizado.

El esquema iterativo es de la forma:

```
R m(T x){
      y = i0(x);
      while(g(y,x)){
            y ≐ s(x,y);
      }
      return h(x,y);
}
```

Partimos de un Invariante $I(x,y)$ que debe cumplirse antes de la sentencia *while*, y a final de cada iteración de la misma. Con esos requisitos el algoritmo garantiza al acabar la postcondición:

$$r = h(x,y), I(x,y), !\, g(x)$$

Dónde r es el valor devuelto por el algoritmo. Además es necesario tener en cuenta una función de cota $C(x,y)$ que tome valores enteros mayores o iguales que cero y cuyo valor disminuya en cada iteración.

Para abordar el primero de los problemas nos planteamos generalizar el problema original (con una sola propiedad n) con las nuevas propiedades i, r. Escogemos el invariante $r = i!$ y la función de cota $C(n,i) = n - i$. A partir de aquí iremos rellenando cada uno de los elementos del algoritmo iterativo. En primer lugar la función $i0$ que debe garantizar que la nuevas variables generalizadas cumplan el invariante. Obtenemos $(r,i) \doteq (1,0)$.

El siguiente paso es encontrar el cuerpo del bucle. Asumimos que la variable i aumentará en 1 para que disminuya la función de cota. Por lo tanto tenemos:

$$\begin{matrix} r' = i'! \\ i' = i + 1 \end{matrix} \equiv r' = (i+1)! = (i+1)i! = (i+1)r$$

Por lo que obtenemos las relaciones entre los valores previos y posteriores de las variables expresadas mediante la asignación paralela:

$$(i,r) = (i+1, (i+1)r)$$

Elegimos $g \equiv i < n$ asumiendo que su contario es $!\, g \equiv i \geq n$. El algoritmo queda:

```
long fac(long n){
      int i;
      int r;
      i = 0;
      r = 1;
      while(i<n){
            i= i+1;
            r = i*r;
      }
      return r;
}
```

Y garantiza $r = i!, i = n$ y por lo tanto $r = n!$. Los detalles para obtener el cuerpo del bucle a partir de la asignación paralela y la conclusión que al final del bucle se cumple $i = n$ (en vez de $i \leq n$) se deja como ejercicio.

Para el segundo problema generalizamos el problema original con las variables a, i. Escogemos el Invariante $a = \sum_{k=0}^{i-1} ls[k]$, asumimos que el tamaño de la lista es n, la función de cota como antes $C(n, i) = n - i$ e incrementamos la variable i en 1 en cada iteración del bucle para conseguir que disminuya la función de cota.

```
double sum(List<Double> ls){
      int i;
      double a;
      int n = ls.size();
      i = 0;
      a = 0.;
      while(i<n){
            a= a + ls[i];
            i = i+1;
      }
      return a;
}
```

La justificar el cuerpo del bucle tenemos que:

$$a = \sum_{k=0}^{i-1} ls[k] \equiv a' = \sum_{k=0}^{i'-1} ls[k] = \sum_{k=0}^{i} ls[k] = ls[i] + \sum_{k=0}^{i-1} ls[k] = ls[i] + a$$
$$i' = i + 1$$

Lo que da lugar a la asignación paralela:

$$(a, i) = (a + ls[i], i + 1)$$

El resto de elementos es similar a la anterior.

En el tercer problema partimos de las listas ordenadas l, s de tamaños respectivos n, m y queremos encontrar una lista ordenada r que contenga los elementos de las dos listas anteriores. Generalizamos el problema con las variables i, j, k y escogemos el invariante que la sublista $r[0, k]$ esté ordenada y contenga los elementos de las sublistas $l[0, i]$ y $s[0, j]$. Del invariante escogido podemos derivar la propiedad $k = i + j$ y una de las siguientes situaciones alternativas:

$$0 < i < n, 0 < j < m, l[i-1] \leq s[j-1], r[k-1] = l[i-1]$$
$$0 < i < n, 0 < j < m, l[i-1] > s[j-1], r[k-1] = s[j-1$$
$$0 < i < n, j = m, r[k-1] = l[i-1]$$
$$i = n, 0 < j < m, r[k-1] = s[j-1]$$
$$i = 0, j = 0, k = 0$$

No vamos a mirar con más detalle las condiciones anteriores pero todas ellas son casos que se deducen del requisito de que sublista $r[0, k]$ esté ordenada y contenga los elementos de las sublistas $l[0, i]$ y $s[0, j]$. A partir de aquí el algoritmo iterativo resulta ser:

```
List sum(List l, List s){
      int i = 0;
      int j = 0;
      int k = 0;
      List r = {};
      int n = l.size();
      int m = s.size();
      while(k<n+m){
            if(i<n && j<m && l[i-1]<=s[i-1]){
                  r[k-1] = l[i-1];
                  i = i+1;
            }
            if(i<n && j<m && l[i-1]>s[i-1]){
                  r[k-1] = s[j-1];
                  j = j+1;
            }
            if(i<n && j==m){
                  r[k-1] = l[j-1];
                  i = i+1;
            }
            if(i==n && j<m){
                  r[k-1] = s[j-1];
                  j = j+1;
            }
            k = k+1;
      }
      return r;
}
```

Se deja como ejercicio comprobar que se cumple el invariante en cada iteración y que la función de cota $C(n, m, i, j, k) \equiv n + m - k$ disminuye en cada iteración.