Tema 3: Recursividad

Tecnología de la Programación

Índice

Índice

Introducción

La Matrioska es un conjunto de muñecas de madera tradicionales rusas. La curiosidad de estas muñecas reside en que están huecas por dentro, y que una muñeca contiene otra muñeca más pequeña dentro de sí; a su vez, esta muñeca también contiene otra muñeca dentro. Y así, una dentro de otra.

Introducción

Recursividad:

- Concepto fundamental en matemáticas e informática
- Técnica de resolución de problemas alternativa a la iteración
  - Buscaremos la técnica más adecuada dependiendo del problema, valorando:
    - Sencillez de la solución
    - Eficiencia de la solución
  - En general, recursividad puede ser más intuitiva pero más ineficiente
- Otras aplicaciones:
  - Definición de TDs
  - Descripción de sintaxis y semántica en lenguajes de programación

Introducción

Definición. Una función se dice recursiva si forma parte de si misma o se define en función de si misma

Ejemplo: Función factorial

$$\begin{cases} n! = n \times (n-1)! \\ 0! = 1 \end{cases} \quad \text{en lugar de} \quad \begin{cases} n! = n \times (n-1) \times \ldots \times 1 \\ 0! = 1 \end{cases}$$

$$fact(n) = \begin{cases} n \times fact(n-1) & \texttt{si } n > 0 \\ 1 & \texttt{si } n = 0 \end{cases}$$

Introducción

Problema: Diseñar un algoritmo para cálculo del factorial

- Si tenemos definida la función recursivamente, codificarla es trivial
- Para poder definir la función, el primer paso es poder definir de forma recursiva el dominio y la imagen

Introducción

Una función queda definida al especificar:

- Nombre
- Dominio = conjunto inicial
- Imagen = conjunto final
- Valor de la imagen que se le asocia a cada elemento del dominio

Ejemplo.

- Nombre: factorial
- Dominio: $\mathbb{N}$
- Imagen: $\mathbb{N}^+$
- $$\begin{array}{rccl} factorial: & \mathbb{N} & \to & \mathbb{N}^+ \\ & 0 & \mapsto & 1 \\ & n & \mapsto & 1 \times 2 \times \ldots \times n \end{array}$$

Introducción

La definición de una función recursiva se realiza en dos etapas:

1. Caso base: se dan las imágenes de los elementos básicos del dominio
2. Caso recursivo (recurrencia): se describen las reglas a partir de las cuales se especifica el valor de la función para un elemento en términos de los valores de la función para otros elementos ya calculados

Ejemplo. Factorial:

- Caso base: factorial(0)=1
- Recurrencia: factorial(n) = n x factorial(n-1)

Introducción

La definición recursiva de la función factorial es posible gracias a que el dominio de la función también puede definirse de manera recursiva

Definición recursiva de conjuntos

- Base: se dan las imágenes de los elementos básicos del dominio
- Recurrencia: se formulan recursivamente reglas que permiten crear un elemento del conjunto a partir de los elementos ya creados

Ejemplo. Números naturales

- Caso base: 0 es un número natural
- Recurrencia: el siguiente de un número natural es un número natural

Introducción

Ejemplo. Dado un alfabeto L = {A,B, ...} definimos las cadenas sobre L = L^*

- Base: $\perp$ (cadena vacía)
- Recurrencia: $c \in L^*$ y $l \in L \Rightarrow cl \in L^*$

Potencia de recursividad:

- definimos conjuntos infinitos a partir de enunciados finitos

Índice

Algoritmos recursivos

Definición. Un algoritmo se dice recursivo cuando en su secuencia de instrucciones aparece una llamada a si mismo

Los algoritmos recursivos son adecuados cuando el problema, la función a calcular o las estructuras de datos están ya definidos de manera recursiva

Algoritmo recursivo → Función recursiva → Dominio recursivo

Algoritmos recursivos

Diseño de algoritmos recursivos requiere dar dos pasos:

1. Caso base: determinar casos triviales; es decir, casos que no necesitan auto referencia
2. Recurrencia: determinar el caso general; es decir, encontrar una ley de recurrencia que defina el comportamiento del algoritmo para unas entradas en términos del comportamiento ya conocido para otras entradas

Importante. Con la ley de recurrencia debemos acercarnos cada vez más a algún caso trivial. En caso contrario → Recursividad mal fundada.
Ej. $n!=n \times (n-1)!$ para $n < 0$

Algoritmos recursivos

Ejemplo: Factorial Recursivo

Algoritmos recursivos: traza

Para hacer la traza de un algoritmo recursivo creamos unas estructuras de datos que llamamos pilas:

- En las pilas se añaden y/o eliminan elementos por un único extremo
- Para hacer la traza se almacenan los valores que van tomando:
  - Variables locales
  - Parámetros formales
  - Dirección de retorno de la función
- Los identificadores son los mismos en todas las llamadas, pero en cada llamada se asocian los valores correspondientes que se depositan en la pila
- Al alcanzar caso base se eliminan de la pila los últimos elementos (cima) y se devuelve control al punto donde se hizo la llamada recursiva

Algoritmos recursivos: coste computacional

Al analizar la eficiencia de un algoritmo recursivo es frecuente la aparición de ecuaciones recursivas denominadas ecuaciones recurrentes

Ejemplo factorial recursivo:

- Complejidad O(n)
- Complejidad en tiempo de factorial recursivo e iterativo es la misma
- Complejidad en espacio, factorial recursivo más ineficiente ya que requiere guardar en cada llamada recursiva el estado de ejecución correspondiente a la llamada anterior

En general, versión recursiva es más ineficiente que la iterativa

Índice

Tipos de recursividad

- Recursividad lineal: cada llamada al algoritmo recursivo genera a lo más una llamada recursiva. Ejemplo: factorial
- Recursividad no lineal (o múltiple): cada llamada al algoritmo recursivo puede generar más de una llamada recursiva.
  - Ejemplo: Fibonacci

$$F(n) = \begin{cases} n & \texttt{si } n = 0, 1 \\ F(n-1) + F(n-2) & \texttt{si } n \geq 2 \end{cases}$$

  - Traza de algoritmos recursivos utilizando pilas sólo es posible en el caso de la recursividad lineal, en caso de recursividad no lineal se usan árboles

Tipos de recursividad

Recursividad lineal:

- Recursividad final: la llamada recursiva es lo último que se ejecuta antes de terminar la ejecución del algoritmo. Ejemplo. `mostrarInv`
- Recursividad no final: después de la llamada recursiva quedan instrucciones por ejecutar. Ejemplo. `mostrar`

Tipos de recursividad

Eficiencia en espacio:

- Algoritmos recursivos finales:
  - Compiladores “se dan cuenta” de que no hay nada más después de la llamada recursiva y no almacenan la llamada en la pila
- Algoritmos recursivos no finales:
  - Más ineficientes porque se almacena la pila de llamadas
  - Solución: convertir algoritmos no finales en finales utilizando inmersión (lo veremos más adelante)

Índice

Búsqueda en un vector

- Búsqueda secuencial en un vector cualquiera
- Búsqueda en un vector ordenado:
  - Búsqueda secuencial
  - Búsqueda dicotómica (o binaria)

Búsqueda secuencial en un vector cualquiera

```
función bsr (tVector v, entero n, entero clave) dev booleano
{Pre: 0≤n≤100, v de tamaño n}
{Post: dev VERDAD si ∃ i con 0≤i<n tq v[i]=clave, y FALSO en caso contrario}
```

Planteamiento: recorrer el vector buscando la `clave`

- Si está ⇒ cuando se encuentra `v[i]=clave` ⇒ VERDAD
- Si no está ⇒ cuando se ha recorrido el vector completo y se ha llegado al final

¿Cómo recorremos el vector? ¿Cómo hacemos la llamada recursiva?

- Iterativo: empezamos desde el principio
- Recursivo: empezar por el final, `n` representa el tamaño del vector (el último elemento, si el vector no es vacío, está en la posición n-1)

Búsqueda secuencial en un vector cualquiera

Resumen: Comenzamos por la última componente

- Si es dato que buscamos ⇒ dev VERDAD
- Si no lo es ⇒ el problema se transforma en buscar en un vector con una componente menos
- Si llegamos a una componente fuera de rango, es porque el dato a buscar no está ⇒ dev FALSO

Importante: Cuidado con el orden de los casos al implementar

Búsqueda secuencial en un vector ordenado

```
función bsrOrd (tVector v, entero n, entero clave) dev booleano
{Pre: 0≤n≤100, v de tamaño n y ordenado de forma creciente}
{Post: dev VERDAD si ∃ i con 0≤i<n tq v[i]=clave, y FALSO en caso contrario}
```

Planteamiento recursivo: igual que antes pero añadiendo otro caso base

- Si está ⇒ parar cuando se encuentra `v[n-1]=clave` ⇒ VERDAD
- Si no está ⇒ parar porque:
  - Se ha recorrido el vector completo → `n=0`
  - Se ha encontrado un dato más pequeño que la clave `v[n-1]<clave`

Búsqueda dicotómica (binaria) en vector ordenado

```
función bdOrd (tVector v, entero n, entero clave) dev booleano
{Pre: 0≤n≤100, v de tamaño n y ordenado de forma creciente}
{Post: dev VERDAD si ∃ i con 0≤i<n tq v[i]=clave, y FALSO en caso contrario}
```

Planteamiento: encontrar punto medio del vector y compararlo con `clave`

- Si coinciden ⇒ dev VERDAD
- Si `clave` es mayor ⇒ buscar en la parte derecha del vector
- Si `clave` es menor ⇒ buscar en la parte izquierda del vector

El proceso se repite pero con un vector más pequeño (hemos reducido el tamaño a la mitad)

Búsqueda dicotómica (binaria) en vector ordenado

Notar que:

- 1ª llamada a la función se efectúa sobre vector completo
- Siguientes llamadas sobre partes del vector ⇒ delimitar parte del vector que queda, ¿cómo?
  - Por medio de dos índices `izq` y `der`

Pasamos de:

- buscar en vector completo ordenado (problema inicial) a
- buscar en un tramo ordenado de un vector

Búsqueda dicotómica (binaria) en vector ordenado

Para poder llamar recursivamente a la función hay que generalizarla ya que los subproblemas a resolver no son exactamente iguales al de partida

Para resolver problema de partida se introduce en otro más general. A este proceso muy utilizado en recursividad se le denomina inmersión

Búsqueda dicotómica (binaria) en vector ordenado

```
función bdOrdInm (tVector v, entero izq, entero der, entero clave) dev booleano
{Pre: v de tamaño n y ordenado crecientemente, 0≤izq, -1≤der<n, e izq≤der+1}
{Post: dev VERDAD si ∃ i con izq≤i≤der tq v[i]=clave, y FALSO en caso contrario}
```

```
función bdOrd (tVector v, entero n, entero clave) dev booleano
{Pre: 0≤n≤100, v de tamaño n y ordenado de forma creciente}
{Post: dev VERDAD si ∃ i con 0≤i<n tq v[i]=clave, y FALSO en caso contrario}
```

Índice

1. Introducción
2. Algoritmos recursivos
3. Tipos de recursividad
4. **Algunos algoritmos recursivos**
    a. Búsqueda en un vector
    b. Ordenación de un vector
5. Recursividad vs Iteración

Algoritmos de ordenación recursiva

Veremos dos algoritmos de ordenación recursiva basados en la técnica divide y vencerás:

- Mergesort
- Quicksort

Técnica divide y vencerás

Técnica de resolución de problemas

Idea intuitiva:

- Descomponer problema original en subproblemas más pequeños
- Resolver cada uno de los subproblemas
- Combinar soluciones de los subproblemas para obtener solución del problema original

```
función divideYvencerás (problema)
principio
     Descomponer el problema en n subproblemas más pequeños
     para i ← i hasta n hacer
          Resolver el subproblema i
     fpara
     Combinar las n soluciones
fin
```

Técnica divide y vencerás

- Si subproblemas son todavía demasiado grandes, aplicar la misma técnica
- Para ello, diseñar algoritmo recursivo que divida el problema hasta un caso trivial

```
función divideYvencerás (problema)
principio
     si problema es trivial entonces
          Resolver el problema
     si_no
          Descomponer el problema en n subproblemas más pequeños
          para i ← i hasta n hacer
               divideYvencerás(subproblema i)
          fpara
          Combinar las n soluciones
     fsi
fin
```

Mergesort

Objetivo: ordenar un vector de enteros de manera creciente

Solución: divide y vencerás:

- Divide el vector por la mitad hasta alcanzar un vector con un único elemento
- Juntar los vectores de manera ordenada

6 5 3 1 8 7 2 4

Mergesort

```
acción mergeSort(e/s tVector v, ent entero n)
{PRE: v de tamaño n; 0≤n≤100}
{POST: las n primeras componentes de v son ordenadas de manera creciente}
principio
     mergeSortInm(v,0,n-1)
fin
```

Mergesort

```
acción mergeSortInm(e/s tVector v, ent entero inf, ent entero sup)
{PRE: v de tamaño ntam; 0≤inf≤sup+1≤n≤100}
{POST: Para todo i y j tq inf≤i≤j≤sup se tiene que v[i]≤v[j]}
variables
     entero m
principio
     si (inf ≤ sup) entonces
          m ← (inf + sup) DIV 2
          mergeSortInm(v,inf,m)
          mergeSortInm(v,m+1,sup)
          combinar(v,inf,m+1,sup)
     fsi
fin
```

Mergesort

En metodología de la programación visteis cómo combinar dos vectores:

- Mientras tengo elementos en los dos vectores, cojo el más pequeño
- Termino uno
- Termino otro

Se puede hacer lo mismo sobre dos tramos del mismo vector, pero para eso necesitamos delimitar los extremos de ambos tramos

Mergesort

```
acción combinar(e/s tVector v, ent entero iz, ent entero iz2,
                ent entero de2)
{PRE: v de tamaño n; 0≤iz1≤iz2≤de2<n, las componentes de v entre iz1 e iz2 están
ordenadas de manera creciente, y lo mismo para las que están entre iz2 y de2}
{POST: componentes de iz1 a de2 ordenadas de manera creciente mezclando componentes
de iz1 a iz2-1 y de iz2 a de2}
variables
    tVector aux
    Entero i, de1, i1, i2, j
```

Combinar

```
principio
     i1 ← iz1; i2 ← iz2; de1 ← iz2-1; i ← 0
     mientras que (i1 ≤ de1 AND i2 ≤ de2) hacer
          si (v[i1]<v[i2]) hacer
               aux[i] ← v[i1]
               i1++
          si_no
               aux[i] ← v[i2]
               i2++
          fsi
          i++
     fmq
     mientras que (i1 ≤ de1) hacer
          aux[i] ← v[i1]
          i1++
          i++
     fmq
     mientras que (i2 ≤ de2) hacer
          aux[i] ← v[i2]
          i2++
          i++
     fmq
     i ← 0
     para j ← iz1 hasta de2 hacer
          v[j] ← aux[i]
          i++
     fpara
fin
```

Mergesort

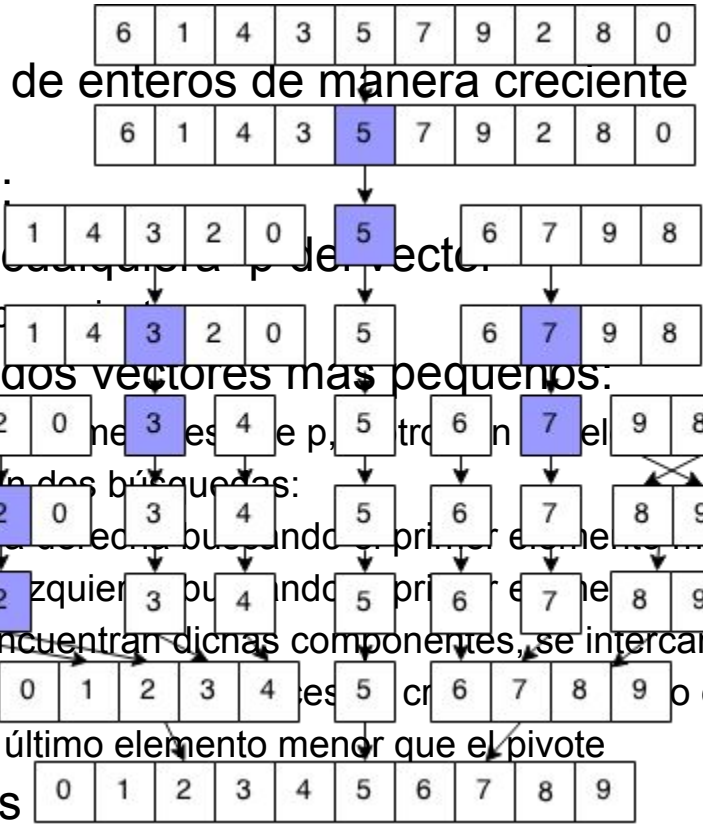
- Si vector de tamaño n, lo podemos dividir a la mitad log(n) veces
- Cada vez que lo dividimos, debemos mezclar n elementos → O(n)
- Por lo tanto la complejidad es O(n log(n))

Quicksort

Objetivo: ordenar un vector de enteros de manera creciente

Solución: divide y vencerás:

- Toma un elemento cualquiera "p" del vector.
  - A p se le conoce como
- Divide el vector en dos vectores mas pequeños:
  - Uno con los elementos menores que p, otro con los mayores que p
  - Para ello se realizan dos búsquedas:
    - De izquierda a derecha buscando el primer elemento mayor que el pivote
    - De derecha a izquierda buscando el primer elemento menor que el pivote
    - Cuando se encuentran dichas componentes, se intercambian
    - Se continua hasta encontrar el punto en el que se intercambia pivote con el último elemento menor que el pivote
- Ordena los vectores



Quicksort

```
acción quickSort(e/s tVector v, ent entero n)
{PRE: v de tamaño n; 0≤n≤100}
{POST: las n primeras componentes de v son ordenadas de manera creciente}
principio
     quickSortInm(v,0,n-1)
fin
```

Quicksort

```
acción quickSortInm(e/s tVector v, ent entero inf, ent entero sup)
{PRE: v de tamaño n; 0≤inf≤sup+1≤n≤100}
{POST: Para todo i y j tq inf≤i≤j≤sup se tiene que v[i]≤v[j]}
variables
     entero m
principio
     si (inf ≤ sup) entonces
          particion(v,inf+1,sup,v[inf],m)
          permutar(v[inf],v[m])
          quickSortInm(v,inf,m-1)
          quickSortInm(v,m+1,sup)
     fsi
fin
```

```
acción particion(e/s tVector v, ent entero iz,
                 ent entero de, ent entero pivote, sal entero pos)
{PRE: v de tamaño n; 0≤iz≤de+1≤n≤100}
{POST: iz≤pos≤de tal que
       ∀ j con iz≤j≤pos, v[j]≤pivote; y ∀ i con pos+1≤i≤de, pivote≤v[i]}
principio
     mientras que (iz ≤ de) hacer
          mientras que (iz ≤ de AND v[iz] ≤ pivote) hacer
               iz++
          fmq
          mientras que (iz ≤ de AND pivote ≤ v[de]) hacer
               de--
          fmq
          si (iz ≤ de) entonces
               permutar(v[iz],v[de])
               iz++
               de--
          fsi
     fmq
     pos ← iz-1
fin
```

Eficiencia algoritmos de ordenación

- Selección directa: $O(n^2)$
- Inserción directa:
  - Mejor caso: $O(n)$
  - Peor caso: $O(n^2)$
- Burbuja: $O(n^2)$
- Heapsort: $O(n \log n)$
- Mergesort: $O(n \log n)$
- Quicksort:
  - Media: $O(n \log n)$
  - Peor caso: $O(n^2)$

Índice

Recursividad vs Iteración

- Técnicas alternativas
- Primera etapa del diseño de algoritmos:
  - Solución clara y correcta
  - En algunos casos la solución recursiva es muy clara
  - Pruebas de corrección se ven facilitadas por el uso de la recursividad
- Solución recursiva suele ser más ineficiente que la iterativa

¿Cuándo usar la recursividad?

- Cuando sea la natural:
  - Dificultad solución iterativa no justifica ganancia en tiempo de ejecución
  - Aproximación recursiva puede dar lugar a algoritmos muy eficientes (e.g. Quicksort)
- Situación ideal:
  - Primer paso: solución recursiva clara
  - Segundo paso: transformar solución recursiva en una iterativa con comportamiento equivalente
- Afortunadamente, transformación siempre posible
  - Lo hacen algunos compiladores de manera automática

Esquema de transformación

Recordatorio:

- Recursividad lineal (e.g. factorial, potencia, …)
  - Recursividad final (e.g. bdr)
  - Recursividad no final (e.g. factorial)
- Recursividad no lineal (e.g. fibonacci)

Existen esquemas de transformación sistemáticos para todos los casos

Nos centramos en los casos de:

- recursividad lineal y final → iterativo
- recursividad no final → recursividad final

Recursividad no final → Recursividad final

- Posible construir un algoritmo con recursión final a partir de uno con recursión no final
- Para ello se utiliza inmersión en un problema más general
- Proceso no siempre trivial
- Hay compiladores capaces de aprovecharse

Recursividad final → iterativo

Pasos:

1. Condición con llamada recursiva → bucle
2. Caso base → condición de parada del bucle
3. Cuerpo del bucle:
    - Sentencias que preceden a la llamada recursiva
    - Transformación de parámetros del algoritmo (acercamiento a caso base)
4. Solución caso base → final algoritmo iterativo y fuera del bucle

Recursividad final → iterativo

```
subalgoritmo Recursivo (tipo x)
principio
    si C(x) entonces
        A1(x)
    si_no
        A2(x)
        Recursivo(T(x))
    fsi
fin
```



```
subalgoritmo Iterativo (tipo x)
principio
    mientras que not C(x) entonces
        A2(x)
        x ← T(x)
    fmq
    A1(x)
fin
```