Universidad Autónoma de Madrid

Escuela Politécnica Superior

Estructuras de Datos

Recursión

Índice

Algoritmo recursivo

Definición:

- Un **algoritmo recursivo** es un algoritmo que resuelve un problema llamándose a sí mismo sobre instancias más simples del problema
- Incluye uno o más **casos base** para los que la solución se obtiene trivialmente sin recursión

Ejemplos:

- Muchas operaciones sobre árboles se realizan recursivamente: inserción y búsqueda sobre un BST, recorrido de un árbol, etc.
- Algunas funciones matemáticas se definen recursivamente: factorial, sucesión de Fibonacci, etc.

Divide y vencerás

- La recursión es útil cuando:
  1. El problema se descompone fácilmente en subproblemas
  2. Cada subproblema es más fácil de resolver que el problema original
  3. La solución al problema original se puede obtener a partir de las soluciones de los subproblemas
- Las funciones recursivas suelen requerir más memoria y recursos computacionales que sus equivalentes iterativos, pero en ocasiones son más sencillas y naturales

Factorial

$$\begin{cases} 0! = 1 \\ n! = n(n-1)! \quad \text{para } n > 0 \end{cases}$$

Factorial

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

```
int factorial(int n) {
  /* Base case: */
  if (n == 0) {
    return 1;
  }

  /* Recursion: */
  return n * factorial(n - 1);
}
```

Sucesión de Fibonacci

$$\begin{cases} F_0 = 0 \\ F_1 = 1 \\ F_n = F_{n-2} + F_{n-1} \quad \text{para } n > 1 \end{cases}$$

Sucesión de Fibonacci

$$\begin{cases} F_0 = 0 \\ F_1 = 1 \\ F_n = F_{n-2} + F_{n-1} \quad \text{para } n > 1 \end{cases}$$

```c
int fibonacci(int n) {
  /* Base cases: */
  if (n == 0) {
    return 0;
  }
  if (n == 1) {
    return 1;
  }

  /* Recursion: */
  return fibonacci(n - 2) + fibonacci(n - 1);
}
```

Búsqueda binaria (I)

Problema: Buscar un elemento `n` en un array **ordenado** `t`

- Devolver el índice del elemento si se encuentra
- Devolver `-1` si no se encuentra

1	2	3	4	5	6	7	8	9

Búsqueda binaria (I)

Problema: Buscar un elemento `n` en un array **ordenado** `t`

- Devolver el índice del elemento si se encuentra
- Devolver `-1` si no se encuentra

1	2	3	4	5	6	7	8	9

Idea: Comparar con el elemento `t[mid]` en la posición central

- Si `e == T[mid]` devolver `mid`
- Si `e < T[mid]` continuar buscando en la mitad izquierda
- Si `e > T[mid]` continuar buscando en la mitad derecha

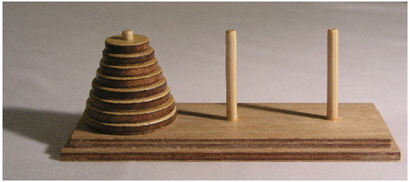
Búsqueda binaria (II)

```
int binSearch(int n, int *t, int low, int high) {
  int mid;

  /* Base case 1, n not found: */
  if (low > high) return -1;

  mid = (low + high) / 2;
  /* Base case 2, n found: */
  if (t[mid] == n) return mid;

  /* Recursion on left part: */
  if (t[mid] > n) return binSearch(n, t, low, mid - 1);

  /* Recursion on right part: */
  else return binSearch(n, t, mid + 1, high);
}
```

Torres de Hanoi (I)

Las Torres de Hanoi es un rompecabezas donde se deben mover todos los discos del primer poste al segundo, con las siguientes reglas:

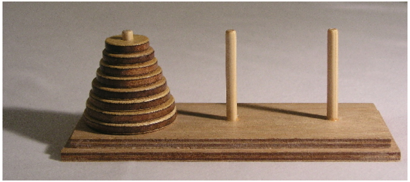
1. Solo se puede mover un disco cada vez
2. Solo se puede mover el disco que esté en lo alto de una pila
3. El disco solo se puede poner encima de una pila diferente o en un poste vacío
4. No se puede colocar nunca un disco encima de otro de menor diámetro



Animación de las Torres de Hanoi: `http://towersofhanoi.info/Animate.aspx`

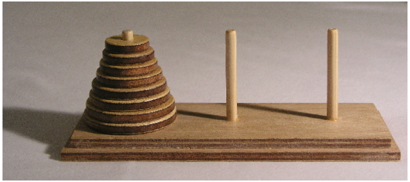
Torres de Hanoi (II)

Algoritmo recursivo: Se puede diseñar una función recursiva para resolver las Torres de Hanoi para n discos



Torres de Hanoi (II)

Algoritmo recursivo: Se puede diseñar una función recursiva para resolver las Torres de Hanoi para `n` discos



- Caso base: Si `n = 1`, basta con mover el disco del poste `1` al `2`
- Recursión:
  - Resolver para los `n - 1` discos superiores, moviéndolos del poste `1` al poste `3`
  - Mover el disco `1` (el de mayor tamaño) del poste `1` al poste `2`
  - Resolver para los `n - 1` discos del poste `3`, moviéndolos al poste `2`

Torres de Hanoi (III)

```c
void hanoi(int n, int rod1, int rod2, int rod3) {
  /* Base case, just 1 disk: */
  if (n == 1) {
    move(rod1, rod2);
    return;
  }

  /* Recursion: */
  hanoi(n - 1, rod1, rod3, rod2);
  move(rod1, rod2);
  hanoi(n - 1, rod3, rod2, rod1);
}
```

```c
void move(int from, int to) {
  /* Just prints the movement: */
  printf("%d --> %d\n", from, to);
}
```

Torres de Hanoi (IV)

Ejemplo: Ejecución para `hanoi(4, 1, 2, 3)`

```
$ ./hanoi 4
1 --> 3
1 --> 2
3 --> 2
1 --> 3
2 --> 1
2 --> 3
1 --> 3
1 --> 2
3 --> 2
3 --> 1
2 --> 1
3 --> 2
1 --> 3
1 --> 2
3 --> 2
```

Ejercicio: ¿Cuántos movimientos son necesarios para resolver las Torres de Hanoi con `n` discos?

Ventajas e inconvenientes de la recursión

Ventajas:

- Programación más sencilla mediante la descomposición del problema en casos más simples
- Código más elegante e interpretable

Inconvenientes:

- Seguir el flujo del programa puede ser difícil
- Demasiadas llamadas a funciones reduce la eficiencia de un algoritmo
- Son costosas en cuanto a memoria
- Algunos lenguajes de programación no permiten llamadas recursivas

Eliminación de la recursión

Idea: Convertir un algoritmo recursivo, ineficiente y costoso en memoria, en un algoritmo iterativo más eficiente

- Simular el mecanismo de llamadas a función y retornos
- Recursión general: complejo
- Recursión de cola: sencillo

Recursión de cola

- Una llamada recursiva de cola es una llamada recursiva tras la que no se realizan más operaciones
- Una función es recursiva de cola si la llamada recursiva es el último comando ejecutado por la función
- Como no hay nada más que hacer tras la llamada recursiva, la recursión de cola es fácil de simular iterativamente
- Solo hay que reasignar los argumentos de la función y saltar al punto de inicio de la función

Eliminación de la recursión de cola

1. Reasignar los argumentos de la función
2. Saltar al punto de inicio de la función usando un comando tipo `goto`
3. Sustituir la llamada a `goto` por un bucle (`for`, `while`) usando el caso base de la recursión como condición de parada

Eliminación de la recursión de cola: Torres de Hanoi (I)

Versión recursiva:

```
void hanoi(int n, int rod1, int rod2, int rod3) {
  /* Base case, just 1 disk: */
  if (n == 1) {
    move(rod1, rod2);
    return;
  }

  /* Recursion: */
  hanoi(n - 1, rod1, rod3, rod2);
  move(rod1, rod2);
  hanoi(n - 1, rod3, rod2, rod1);
}
```

Eliminación de la recursión de cola: Torres de Hanoi (II)

Versión iterativa con `goto`:

```
void hanoi_iter_1(int n, int rod1, int rod2, int rod3) {
START:
  /* Base case, just 1 disk: */
  if (n == 1) {
    move(rod1, rod2);
    return;
  }

  /* Recursion: */
  hanoi_iter_1(n - 1, rod1, rod3, rod2);
  move(rod1, rod2);

  /* Simulation of tail recursion: */
  swap(&rod1, &rod3);
  n--;
  goto START;
}
```

Eliminación de la recursión de cola: Torres de Hanoi (III)

Versión iterativa con bucle `while`:

```
void hanoi_iter_2(int n, int rod1, int rod2, int rod3) {
  while (n != 1) {
    /* Recursion: */
    hanoi_iter_1(n - 1, rod1, rod3, rod2);
    move(rod1, rod2);

    /* Reassignment of arguments: */
    swap(&rod1, &rod3);
    n--;
  }
  /* Base case, just 1 disk: */
  move(rod1, rod2);
}
```

Impresión de lista (I)

Ejercicio: Detectar la recursión de cola en la siguiente función, y eliminarla en el caso de que exista

```c
Status list_print_rec_1(Node *pn) {
  if (pn == NULL) {
    return OK;
  }
  if (element_print(pn->info) == -1) {
    return ERROR;
  }

  return list_print_rec_1(pn->next);
}
```

Impresión de lista (II)

Ejercicio: Detectar la recursión de cola en la siguiente función, y eliminarla en el caso de que exista

```
int list_print_rec_2(Node *pn) {
  if (pn == NULL) {
    return 0;
  }

  return element_print(pn->info) + list_print_rec_2(pn->next);
}
```

Búsqueda binaria

Ejercicio: Detectar la recursión de cola en la siguiente función, y eliminarla en el caso de que exista

```c
int binSearch(int n, int *t, int low, int high) {
  int mid;

  /* Base case 1, n not found: */
  if (low > high) return -1;

  mid = (low + high) / 2;
  /* Base case 2, n found: */
  if (t[mid] == n) return mid;

  /* Recursion on left part: */
  if (t[mid] > n) return binSearch(n, t, low, mid - 1);

  /* Recursion on right part: */
  else return binSearch(n, t, mid + 1, high);
}
```

Función de Ackermann

Ejercicio: Detectar la recursión de cola en la siguiente función, y eliminarla en el caso de que exista

```
int f(int m, int n) {
  if (m == 0)
    return n + 1;
  else if (n == 0)
    return f(m - 1, 1);
  else
    return f(m - 1, f(m, n - 1));
}
```

Otras funciones recursivas

- **Ejercicio 1:** Detectar la recursión, y eliminarla en el caso de que exista, en las funciones para recorrer los BT en preorden, inorden y postorden
- **Ejercicio 2:** Detectar la recursión, y eliminarla en el caso de que exista, en la función que cuenta el número de nodos de un BT
- **Ejercicio 3:** Dado el siguiente algoritmo para calcular el máximo común divisor (mcd) de dos enteros x e y:

$$\mathrm{mcd}(x,y) = \begin{cases} y & \text{si } y \leq x, \mathrm{mod}(x,y) = 0 \\ \mathrm{mcd}(y,x) & \text{si } x < y \\ \mathrm{mcd}(y, \mathrm{mod}(x,y)) & \text{en otro caso} \end{cases}$$

1. Escribir una función recursiva en C que calcule el mcd de dos enteros
2. Detectar la recursión de cola en la función anterior, y eliminarla en el caso de que exista