Recursividad

Dr. Said P. Martagón

Introducción

- Un procedimiento o función recursiva es aquella que se llama a si misma.
  - La ejecución del proceso recursivo se repite con valores (parámetros) diferentes.

Introducción

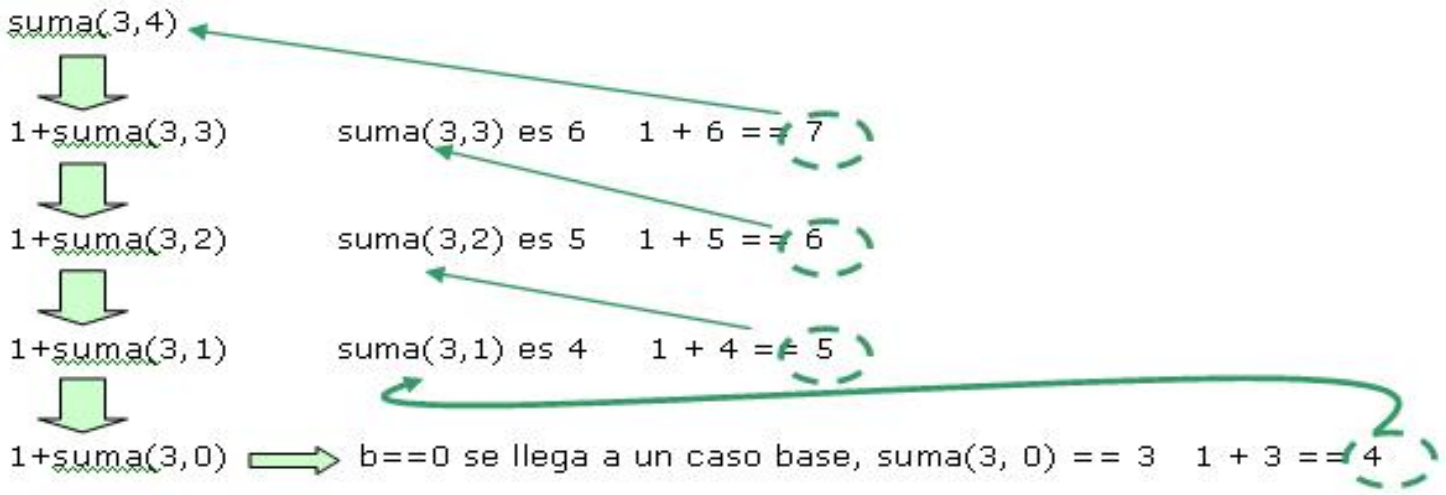
- La recursividad es una alternativa a la iteración muy elegante en la resolución de problemas de naturaleza recursiva. Permite especificar una solución simple y natural para resolver problemas definidos en términos de sí mismos.
  - Ejemplo: Los números naturales
    - 0 es un número Natural
    - El siguiente número de un número natural es otro número natural

Algoritmos Recursivos

- La recursividad está relacionada con el principio de inducción.
  - Existe un caso base, en el que no existe ninguna llamada recursiva.(Condición o criterio base)
  - Existe un caso general conocido como caso inductivo, en las que se realizan llamadas a versiones de la misma función con parámetros diferentes que conducen al caso base.

Algoritmos Recursivos

- Por lo tanto
  - Hay que incluir por lo menos un caso base, que se resuelva sin necesidad de recursividad.
  - Todas las llamadas recursivas deben levar hacia el caso base
- El método debe comprobar si se debe realizar una nueva llamada recursiva o si ya se ha alcanzado el caso base.



Algoritmos Recursivos

- El caso base
  - Supone el final de las llamadas recursivas.
  - Y la realización de llamadas recursiva que lleva a él lo que evita se entre en ciclos infinitos.

Algoritmos Recursivos

- Para crear una función recursiva, es necesario tener una definición recursiva del problema.
  - Ejemplo: Suma de los primeros números naturales.

    Caso Base: s(1) = 1;

    Caso general: s(n) = s(n-1) + n

    El problema esta definido en forma recursiva, para conocer la suma de *n* números se debe conocer previamente la suma de los *n-1* números anteriores.

Suma recursiva de los n primeros números naturales

```
int  SumaNat(int n)
{
    int s;
    if (n = = 1)
          s = 1; // Caso Base
    else
           s = SumaNat(n-1) + n; // Caso general
      return(s);
}
```

Funcionamiento de la recursividad

- Nótese que en una *función recursiva* es necesaria una *condición* para distinguir el caso base del inductivo.
- Para entender como funciona la recursividad es necesario tener bien claro que en memoria no existe una sola versión de la función recursiva.
- Cada vez que se invoque la función recursiva se crea una nueva versión de la misma.
- La estructura de todas la versiones es la misma, pero no así los datos que contiene cada una

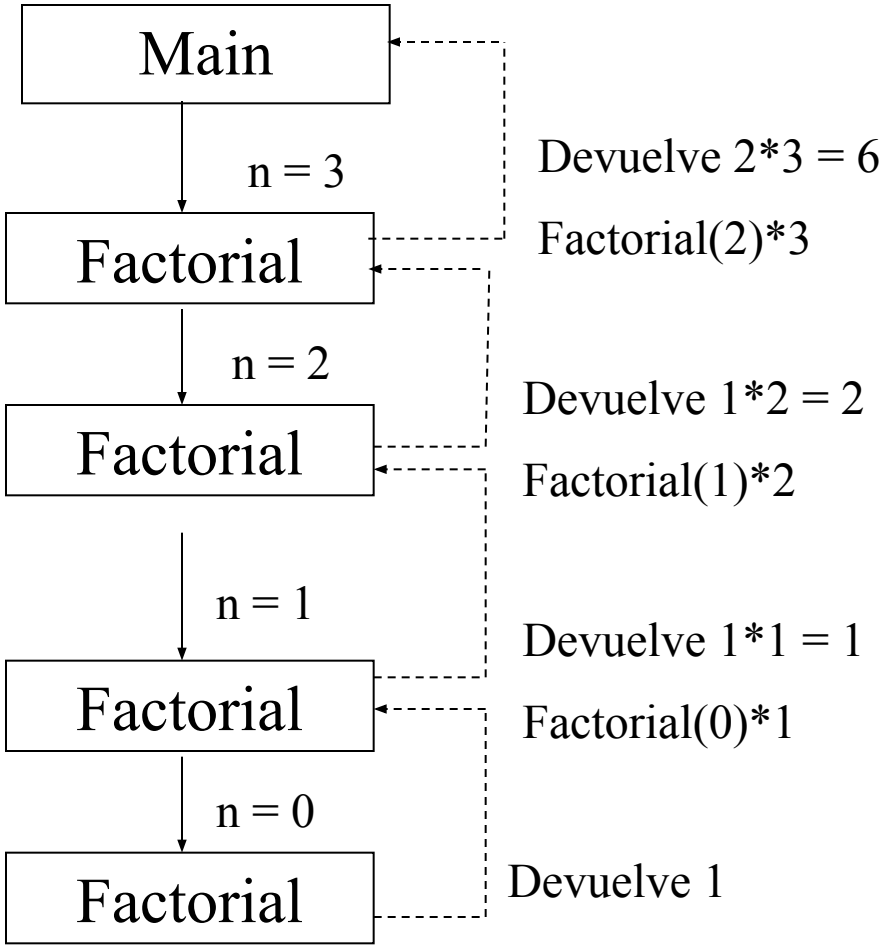
Factorial de un número

Caso base: fact(0) = 1

Caso general: fact(n) = fact(n-1)·n

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

Funcionamiento de la recursividad



IDA VUELTA

Llamada	n	Valor
1ª	3	6
2ª	2	2
3ª	1	1
4ª caso base	0	1

Funcionamiento de la recursividad

- En el anterior ejemplo se ilustro cómo las llamadas recursivas se van produciendo hasta alcanzar el caso base.
- En ese punto se acaban las llamadas y empiezan las devoluciones de valores hasta llegar al método *main*.
- Tenemos un movimiento en 2 sentidos
  1º Hacia delante hasta alcanzar el caso base.
  2º Hacia atrás devolviendo los resultados de cada llamada a la función.
- Las llamadas realizadas implican una estructura pila.
- En cada llamada se realiza una copia de la función recursiva (cada llamada implica una nueva copia de las variables de la función). Esto consume memoria.

Correctitud en la recursividad

- ¿Cómo podemos determinar si un Algoritmo Recursivo es o no correcto?
  - Por simple observación es difícil.
  - Es posible alcanzar ese objetivo con la ayuda del principio de inducción.
    - Verificando 1º el caso base, comprobando si devuelve el resultado correcto para el valor más pequeño.
    - Verificando si el algoritmo funciona correctamente para cualquier valor.

Recursividad vs. Iteración

- Características comunes:
  1. Ambas implican repetición
     - La iteración usa explícitamente una estructura de repetición mientras que la recursión logra la repetición mediante llamadas sucesivas a una función.
  2. Ambas requieren de una condición de fin.
     - La iteración termina cuando deja de cumplirse la condición para terminar el ciclo y la recursión cuando se reconoce un caso base.

Recursividad vs. Iteración

3. Ambas se aproximan gradualmente a la terminación.
   - La iteración continua modificando un contador, hasta que éste adquiere un valor que hace que deje de cumplirse la condición del ciclo.
   - La recursividad sigue produciendo versiones más sencillas del problema original hasta llegar al caso base.

Recursividad vs. Iteración

4. Pueden continuar indefinidamente
   - En la iteración ocurre un ciclo infinito, si la condición del ciclo nunca deja de cumplirse.
   - Se tiene una recursión infinita si cada llamada recursiva no simplifica el problema y no se alcanza el caso base o si aún dirigiéndonos al caso base, lo saltamos.

Recursividad vs. Iteración

- Diferencias.
  - La recursividad presenta una desventaja frente a la iteración: la invocación repetida de la función.Cada llamada hace que se cree otra copia de la función esto puede consumir una cantidad excesiva de memoria.
  - La iteración ocurre en la misma función, con lo que se omite el gasto extra de llamadas a la función.
- Toda tarea que pueda realizarse con recursividad puede también realizarse con una solución iterativa.
- Se elige la solución recursiva cuando este enfoque refleja de forma más natural la solución del problema y produce un programa más fácil de entender y depurar.
- Existen problemas cuya solución iterativa no es viable por lo tanto la recursión es una solución.

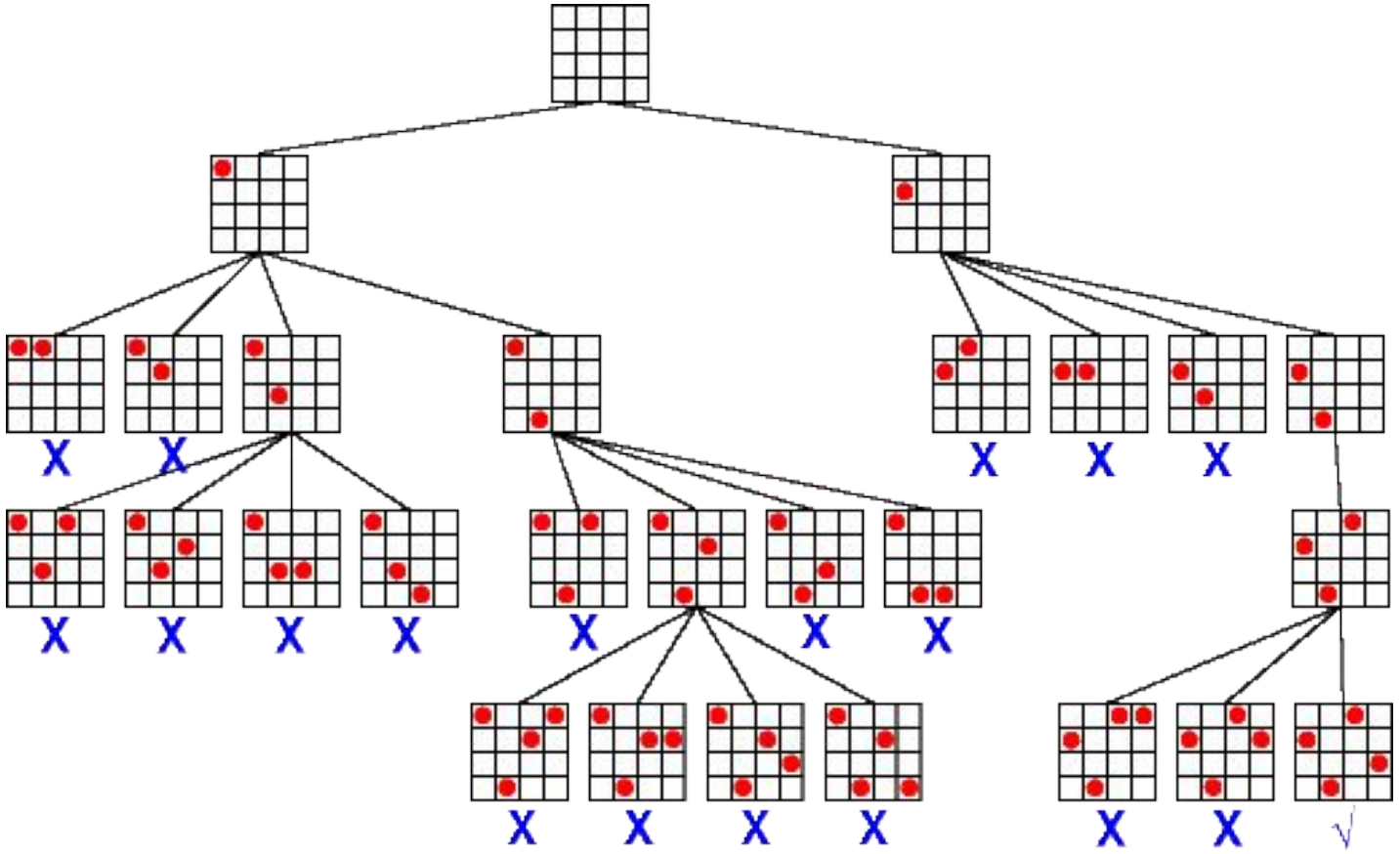
Simulación de la recursividad

- Es posible simular la recursividad a través del uso de una estructura pila y así emular llamadas recursivas.
  - Los parámetros de las llamadas a la función se van almacenado en una pila hasta alcanzar el caso base.
  - La vuelta atrás se consigue, sacando de la pila las los elementos almacenados y procesándolos uno a uno hasta que la pila quede vacía.

Aplicaciones de Recursividad

- Los algoritmos recursivos son muy importantes en el diseño de algoritmos:
  - Backtracking (vuelta atrás), búsqueda exhaustiva, usa recursividad para probar todas las soluciones posibles.
  - Divide y vencerás, transforma el problema de tamaño n en problemas más pequeños de tamaño menor que n.De tal modo que en base a problemas unitarios se construye fácilmente una solución del problema completo. Ej. Búsqueda binaria, ordenamiento Quicksort, Torres de Hanoi.

BackTrack Algorithm



Quicksort

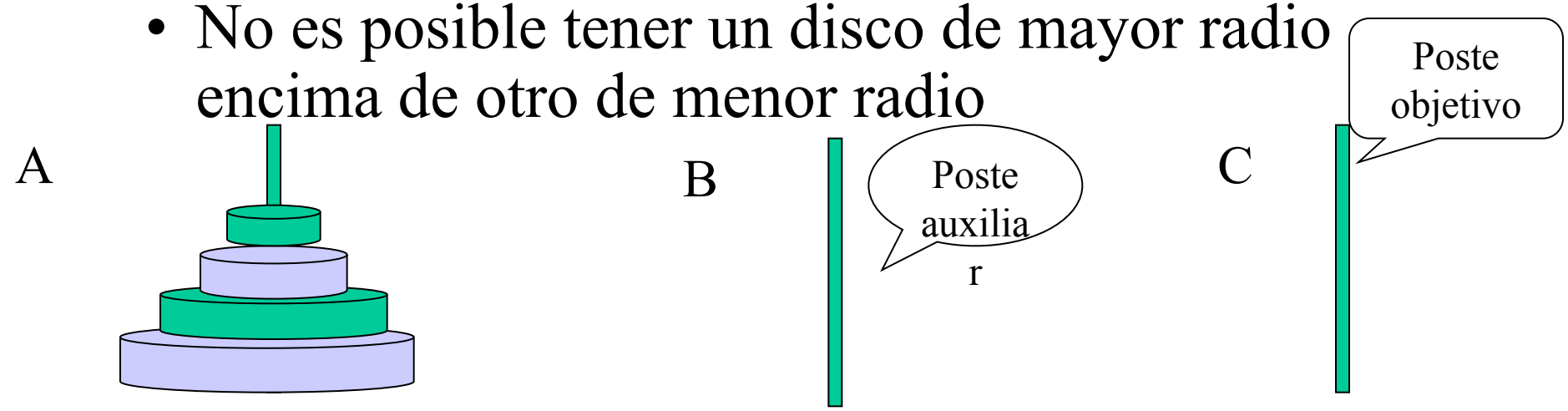
6 5 3 1 8 7 2 4

Torres de Hanoi

- Dice la leyenda que, al crear el mundo, Dios situó sobre la Tierra tres varillas de diamante y sesenta y cuatro discos de oro. Los discos son todos de diferente tamaño e inicialmente fueron colocados en orden decreciente de diámetros sobre la primera de las varillas. También creó Dios un monasterio cuyos monjes tienen la tarea de trasladar todos los discos desde la primera varilla a la tercera. La única operación permitida es mover un disco de una varilla a otra cualquiera, pero con la condición de que no se puede situar encima de un disco otro de diámetro mayor. La leyenda dice también que cuando los monjes terminen su tarea, el mundo se acabará.

Torres de Hanoi (def. problema)

- Se tienen 3 postes A, B y C; en el poste A se tiene *n* discos de tamaño decreciente.
- El objetivo es mover uno a uno los discos desde el poste A al poste C utilizando el poste B como auxiliar.
- No es posible tener un disco de mayor radio encima de otro de menor radio



Torres de Hanoi (solución)

- Mover *n* discos
  - Mover los *n-1* discos superiores de A a B.
  - Mover el disco *n* de A a C.
  - Mover los *n-1* discos de B a C.
- El problema de mover *n* discos se ha transforma en un problema de tamaño *n-1*.
- Mover *n-1* discos
  - Mover los *n-2* discos superiores de A a C.
  - Mover el disco *n-1* de A a B.
  - Mover los *n-2* discos de C a B.
- De este modo se va reduciendo cada vez un nivel la dificultad del problema hasta que el mismo sólo consista en mover un disco.

Torres de Hanoi (solución)

- La técnica consiste en ir intercambiando la finalidad de los postes, origen destino y auxiliar.
- La condición de terminación es que el número de discos sea 1.
- Cada acción de mover un disco realiza los mismos pasos, por lo que es posible expresar la función de manera recursiva.

Torres de Hanoi (solución)

```
void Hanoi (int n, char origen, char destino, char aux)
{
    if (n==1) //Caso básico
        cout<<"\n mover el disco del poste"<<origen<<"  a "<<destino;
    else  /* Caso general: divide y vencerás*/
    {
        Hanoi (n-1, origen, aux, destino);
        cout<<"\n mover el disco del poste"<<origen<<"  a "<<destino;
        Hanoi (n-1, aux, destino, origen);
      }
}
```