Estructura de Datos y Algoritmos 1

Teórico #3:

Recursividad.

Diseño de algoritmos recursivos

Introducción: Recursión o Recursividad

¿Qué significa recurrente?

- Que vuelve a ocurrir (**se repite**) cada cierto tiempo.
- Recurrir a algo: Hacer uso nuevamente

Subprogramas recurrentes

- se invocan (llaman) a sí mismos
- definidos en términos de sí mismos

Introducción: Recursión o Recursividad

La **recursividad** es la propiedad que posee toda función de **invocarse a sí misma** en cualquier parte de su cuerpo de instrucciones.

Ejemplo:

```
void f()
{
      cout << "Llamada recursiva" <<endl;
      f();
}
```

¿Logra terminar correctamente la función **f**?

Introducción: Recursión o Recursividad

Toda función recursiva, debe contener una **condición de parada**, que evite que la función se invoque un número infinito de veces y provoque un error de ejecución que indica que no hay más memoria (***stack overflow***)

Ejemplo:

```
void f(int n)
{
      if (n > 0)        //Condición de parada
      {
            cout << "Llamada recursiva" <<endl;
            f(n-1);
      }
}
```

¿Qué es *stack overflow*? ¿Por qué ocurre?

- cada vez que un subprograma P llama a otro Q debe guardarse una indicación del punto en P donde el control debe retornar al finalizar la ejecución de Q
- Hay una estructura de datos donde se almacenan los sucesivos puntos de retorno.
- Teniendo P → Q_1 → Q_2 → Q_3 → ... → Q_n donde Q_n es el que se está ejecutando, paralelamente se ha formado la estructura de "puntos de retorno"

  p_0 → punto de retorno en P

  p_1 → punto de retorno en Q_1

  …

  P_{n-1} → punto de retorno en Q_{n-1}

¿Qué es *stack overflow*? ¿Por qué ocurre?

p_{n-1}
p_{n-2}
…
…
p_1
p_0

- La estructura **crece** con cada nueva llamada (un lugar) y **decrece** al terminar la ejecución de un subprograma. Se comporta como una PILA (***stack***)
- El tope de la pila es el punto donde debe retornarse el control tras la terminación del subprograma corriente
- si el subprograma corriente llama a otro, el correspondiente punto de retorno debe colocarse como nuevo tope de la pila.
- al finalizar un subprograma, se usa el tope como dirección de retorno y se lo remueve de la pila

Invocaciones

P	Q1	Q2	Q3
---	---	---	---
---	Cód-3	---	---
Cód-1	---	Cód-5	---
---	call Q2	---	Cód-7
call Q1	---	---	---
---	---	call Q3	---
Cód-2	Cód-4	---	---
---	---	Cód-6	---

Stack

Invocaciones

P	Q1	Q2	Q3
---	---	---	---
---	Cód-3	---	---
Cód-1	---	Cód-5	---
---	call Q2	---	Cód-7
call Q1	---	---	---
p_0 : ---	---	call Q3	---
Cód-2	Cód-4	---	---
---	---	Cód-6	---

Stack
p_0

Ejecución: Cód-1

Invocaciones

P	Q1	Q2	Q3
---	---	---	---
---	Cód-3	---	---
Cód-1	---	Cód-5	---
---	call Q2	---	Cód-7
call Q1	p_1 : ---	---	---
p_0 : ---	---	call Q3	---
Cód-2	Cód-4	---	---
---	---	Cód-6	---

Stack
p_1
p_0

Ejecución: Cód-1, Cód-3

Invocaciones

P	Q1	Q2	Q3
---	---	---	---
---	Cód-3	---	---
Cód-1	---	Cód-5	---
---	call Q2	---	Cód-7
call Q1	p_1 : ---	---	---
p_0 : ---	---	call Q3	---
Cód-2	Cód-4	p_2 : ---	---
---	---	Cód-6	---

Stack
p_2
p_1
p_0

Ejecución: Cód-1, Cód-3, Cód-5

Invocaciones

P	Q1	Q2	Q3
---	---	---	---
---	Cód-3	---	---
Cód-1	---	Cód-5	---
---	call Q2	---	Cód-7
call Q1	p_1 : ---	---	---
p_0 : ---	---	call Q3	---
Cód-2	Cód-4	p_2 : ---	---
---	---	Cód-6	---

Stack
p_2
p_1
p_0

Ejecución: Cód-1, Cód-3, Cód-5, Cód-7

Invocaciones

P	Q1	Q2	Q3
---	---	---	---
---	Cód-3	---	---
Cód-1	---	Cód-5	---
---	call Q2	---	Cód-7
call Q1	p_1 : ---	---	---
p_0 : ---	---	call Q3	---
Cód-2	Cód-4	p_2 : ---	---
---	---	Cód-6	---

Stack
p_1
p_0

Ejecución: Cód-1, Cód-3, Cód-5, Cód-7, Cód-6

Invocaciones

P	Q1	Q2	Q3
---	---	---	---
---	Cód-3	---	---
Cód-1	---	Cód-5	---
---	call Q2	---	Cód-7
call Q1	p_1 : ---	---	---
p_0 : ---	---	call Q3	---
Cód-2	Cód-4	p_2 : ---	---
---	---	Cód-6	---

Stack
p_0

Ejecución: Cód-1, Cód-3, Cód-5, Cód-7, Cód-6, Cód-4

Invocaciones

P	Q1	Q2	Q3
---	---	---	---
---	Cód-3	---	---
Cód-1	---	Cód-5	---
---	call Q2	---	Cód-7
call Q1	p_1 : ---	---	---
p_0 : ---	---	call Q3	---
Cód-2	Cód-4	p_2 : ---	---
---	---	Cód-6	---

Stack

Ejecución: Cód-1, Cód-3, Cód-5, Cód-7, Cód-6, Cód-4, Cód-2

Invocaciones

P	Q1	Q2	Q3
---	---	---	---
---	Cód-3	---	---
Cód-1	---	Cód-5	---
---	call Q2	---	Cód-7
call Q1	p_1 : ---	---	---
p_0 : ---	---	call Q3	---
Cód-2	Cód-4	p_2 : ---	---
---	---	Cód-6	---

Stack

Ejecución: Cód-1, Cód-3, Cód-5, Cód-7, Cód-6, Cód-4, Cód-2

Fin de la ejecución de P.

Stack vacío

¿Qué es *stack overflow*? ¿Por qué ocurre?

La pila se hace crecer infinitamente, pero (la memoria de) la máquina es finita, por lo tanto, en algún momento no hay más memoria

(***stack overflow*** = desbordamiento de pila)

Esto ocurre cuando hay **recurrencias infinitas**.

Por eso los subprogramas o algoritmos recursivos precisan **condiciones de parada** o **casos base**.

Ejemplo: Factorial de un número natural

$$n! = n \cdot (n-1) \cdot \cdots \cdot 2 \cdot 1$$

$$0! = 1 \qquad n! = n \cdot (n-1)!$$

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

```
int Fact(int n){
  int fact = 1;
  for(int i=2; i<=n; i++)
    fact *= i;
  return fact;
}
```

- La versión recurrente es más simple, análoga a la definición matemática.

- La versión iterativa es más eficiente (no usa el stack)

A ver si se ha entendido esta intro…

```
void P(int x){
  if (x == 0) cout << x;
  else {
    cout << x;
    P(x-1);
    cout << -1*x;
  }
}
```

El llamado a P(3) ¿qué salida produce?

A ver si se ha entendido esta intro…

```
void P(int x){
  if (x == 0) cout << x;
  else {
    cout << x;
    P(x-1);
    cout << -1*x;
  }
}
```

Stack
cout << -1*x; //x=3

- Llamados: P(3) → P(2)
- Se imprime: 3

A ver si se ha entendido esta intro…

```
void P(int x){
  if (x == 0) cout << x;
  else {
    cout << x;
    P(x-1);
    cout << -1*x;
  }
}
```

Stack
`cout << -1*x; //x=2`
`cout << -1*x; //x=3`

- Llamados: P(3) → P(2) → P(1)
- Se imprime: 3, 2

A ver si se ha entendido esta intro…

```
void P(int x){
  if (x == 0) cout << x;
  else {
    cout << x;
    P(x-1);
    cout << -1*x;
  }
}
```

Stack
`cout << -1*x; //x=1`
`cout << -1*x; //x=2`
`cout << -1*x; //x=3`

- Llamados: P(3) → P(2) → P(1) → P(0)
- Se imprime: 3, 2, 1, 0

A ver si se ha entendido esta intro…

```
void P(int x){
  if (x == 0) cout << x;
  else {
    cout << x;
    P(x-1);
    cout << -1*x;
  }
}
```

Stack
cout << -1*x; //x=2
cout << -1*x; //x=3

- Se imprime: 3, 2, 1, 0, -1

A ver si se ha entendido esta intro…

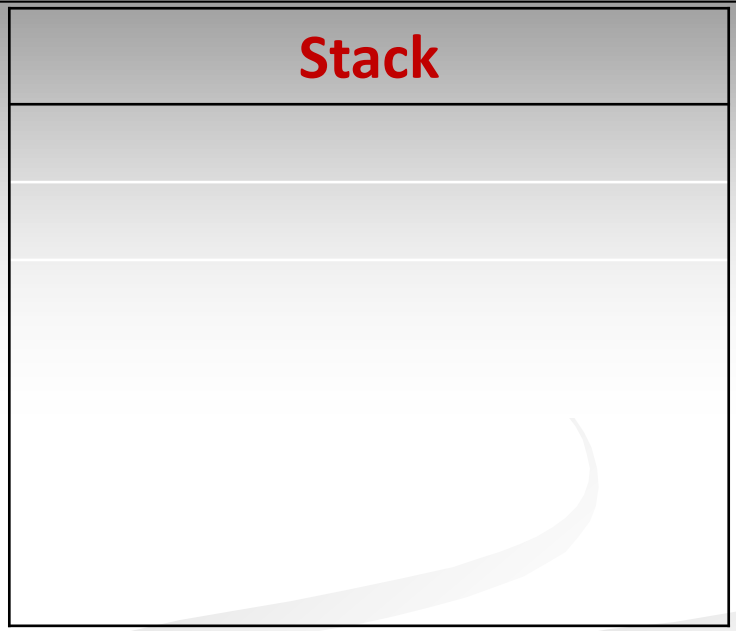
```
void P(int x){
  if (x == 0) cout << x;
  else {
    cout << x;
    P(x-1);
    cout << -1*x;
  }
}
```

Stack
`cout << -1*x; //x=3`

- Se imprime: 3, 2, 1, 0, -1, -2

A ver si se ha entendido esta intro…

```
void P(int x){
  if (x == 0) cout << x;
  else {
    cout << x;
    P(x-1);
    cout << -1*x;
  }
}
```



- Se imprime: 3, 2, 1, 0, -1, -2, -3

FIN

Ejemplo: Sucesión de Fibbonnacci

$0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots$

La sucesión de Fibbonnacci se define como:

$$f_0 = 0$$
$$f_1 = 1$$
$$f_n = f_{n-1} + f_{n-2}, n \geq 2$$

```
int Fibbonacci(int n)
{
  if (n == 0)       //Caso base 1
    return 0;
  else if (n == 1) //Caso base 2
    return 1;
  else              //Caso Inductivo o Recurrente
    return Fibbonacci(n-1) + Fibbonacci(n-2);
}
```

Ejemplo: Máximo Común Divisor

El máximo común divisor (**mcd**) de a y b, $(a > b \geq 0)$, es igual a a si b es cero, en otro caso es igual al **mcd** de b y el resto de a dividido por b

$$mcd(a,b) = \begin{cases} a & \text{si } b = 0 \\ mcd(b, a \bmod b) & \text{en otro caso} \end{cases}$$

```
int mcd(int a, int b)
{
  if (b == 0)         //Caso base 1
    return a;
  else               //Caso Inductivo o Recurrente
    return mcd(b, a % b);
}
```

Ventajas de la recursividad

- Programas más legibles y sencillos.
- Facilidad de programación. Existen problemas para los cuales los algoritmos más intuitivos son recursivos, mientras que su solución iterativa es muy difícil de encontrar.

Desventajas de la recursividad

- Toda llamada recursiva emplea un espacio de memoria que no se puede ignorar al almacenar parámetros y variables locales. Estas necesidades de memoria se multiplican con el número de llamadas recursivas.
- Además de la penalización anterior, algunas funciones recursivas realizan muchos cálculos redundantes, o sea, el mismo cálculo repetidas veces. Ejemplo: Fibonacci.

Caso Base y Caso Recurrente

Caso Base

- Es el cimiento sobre el cual se construirá la solución completa del problema.
- Actúa como condición de terminación de la recursividad. Sin la existencia o identificación del caso base la rutina se llamaría indefinidamente.
- Puede haber más de un caso base

Caso Base y Caso Recurrente

Caso Recurrente

- La parte recurrente es la que se encargará de dividir el problema grande en uno más pequeño hasta llegar a una solución directa o sencilla que será uno de los casos base.
- Es de suma importancia que la parte recurrente converja a un caso base para lograr la terminación del algoritmo.

¿Qué debemos tener en cuenta para definir un algoritmo recursivo?

1. Obtener una definición exacta del problema a resolver, es vital que el problema sea bien definido para no desechar soluciones (aunque, por supuesto, este el primer paso de resolución de cualquier problema de programación).
2. Determinar el tamaño del problema completo que hay que resolver. Este tamaño determinará los valores de entrada en la llamada inicial del algoritmo.
3. Definir los casos bases en los cuales el problema puede expresarse no recursivamente y que pondrá fin a la recursión.
4. Definir el caso general o recurrente en términos de un caso más pequeño del mismo problema, es decir, una llamada recursiva.

¿Qué debemos tener en cuenta para definir un algoritmo recursivo?

1. ¿Cómo se puede definir el problema en términos de uno o más problemas pequeños del mismo tipo del original?
2. ¿Qué instancias del problema harán de caso base?
3. A medida que se reduce el tamaño del problema ¿se alcanzará el caso base?
4. ¿Cómo se usa la solución del caso base para construir una solución correcta del problema original?

Ejemplo: Problema de los contrapesos

Supongamos que tenemos dos cuerpos de pesos desconocidos (valores enteros en kg) y queremos determinar la cantidad de contrapesos que hay que adicionar para equilibrar sus pesos contando con la ayuda de una balanza y de contrapesos de 1kg.

Algoritmo Recursivo

1. Si al colocar los dos cuerpos la balanza permanece equilibrada no necesita contrapeso (0 contrapesos).
2. Si al colocar los dos cuerpos la balanza se desequilibra se realiza una nueva pesada adicionando un contrapeso a la balanza convenientemente (1 + contrapesos necesarios para equilibrar los cuerpos adicionando un contrapeso convenientemente).

Ejemplo: Problema de los contrapesos

1. ¿Cómo se puede definir el problema en términos de uno o más problemas pequeños del mismo tipo del original?

$$Contrapesos(a,b) = \begin{cases} 1 + Contrapesos(a+1,b) & b > a \\ 1 + Contrapesos(a,b+1) & b < a \end{cases}$$

2. ¿Qué instancias del problema harán de caso base?

$$Contrapesos(a,b) = 0 \quad \text{si } a = b$$

Ejemplo: Problema de los contrapesos

3. A medida que se reduce el tamaño del problema ¿se alcanzará el caso base?

   En cada llamada se reduce el tamaño del problema equilibrándose más la balanza por lo que convergerá al caso base

4. ¿Cómo se usa la solución del caso base para construir una solución correcta del problema original?

   $Contrapesos(2,1) = 1 + Contrapesos(2,2) = 1 + 0 = 1$

   Se construye la solución para a=2, b=1 a partir del caso base.

Tipos de recursividades

Recursión simple: Es aquella en cuya definición solo aparece una llamada recursiva. Se puede transformar con facilidad en algoritmos iterativos.

- *Recursión Final:* Si la llamada recursiva es la última operación que se efectúa, devolviéndose como resultado lo que se haya obtenido de la llamada recursiva sin modificación alguna. Ejemplo, cálculo del máximo común divisor.
- *Recursión no final:* El resultado obtenido de la llamada recursiva se combina para dar lugar al resultado de la función que realiza la llamada. Ejemplos, algoritmo para el cálculo del factorial, Problema de la Balanza.

Tipos de recursividades

Recursividad múltiple: Es aquella en la cual hay más de una llamada a sí mismo, dentro del cuerpo de la función. Es mucho más difícil de llevar a la forma iterativa. Ejemplo, el cálculo del *n-ésimo* número de Fibonacci.

Tipos de recursividades

Recursividad cruzada o indirecta: Son algoritmos donde una función provoca una llamada a sí misma de forma indirecta, a través de otras funciones. Es decir, es aquella en la que una función llama a otra y esta a su vez llama a la función que la invocó directa o indirectamente

```
bool par(int n){
  if (n == 0)
    return true;

  if (n > 0)
    return impar(n - 1);

  return impar(n + 1);
}
```

```
bool impar(int n){
  if (n == 0)
    return false;

  if (n > 0)
    return par(n - 1);

  return par(n + 1);
}
```

Recursión estructural en listas

Vamos a definir el conjunto de las listas secuenciales finitas de naturales Inductivamente:

Regla 1: Lista vacía

$$\frac{}{[\,] : \text{Lista}}$$

Regla 2: Lista no vacía

$$\frac{n:N \quad L: \text{Lista}}{n.L: \text{Lista}}$$

Regla 3: Esas son todas las listas

Ejemplos:

[ ]

1.[] ([1])

3.1.[] ([3.1])

Recursión estructural en listas

$$f : \text{Lista} \rightarrow \ldots$$
$$f([\,]) = \ldots$$
$$f(x.L) = \ldots f(L)$$

Ejemplo: Largo de una lista

$$largo : \text{Lista} \rightarrow \mathbb{N}$$
$$largo([\,]) = 0$$
$$largo(x.L) = 1 + largo(L)$$

Ejercicios: Recursión estructural en listas

1. Chequear si un elemento está en una lista.
2. Eliminar la primera ocurrencia de un elemento de una lista.
3. Eliminar todas las ocurrencias de un elemento de una lista.
4. Insertar de manera ordenada un elemento en una lista ordenada.
5. Ordenar una lista de menor a mayor usando la función previa (inserción en lista ordenada)
6. Eliminar duplicados de una lista, dejando solo una ocurrencia (la primera) de cada elemento diferente.
7. El algoritmo select sort.

Ejercicios: Recursión estructural en listas

1. Chequear si un elemento está en una lista.

$$pertenece: \mathbb{N} \times \text{Lista} \rightarrow \{true, false\}$$

$$pertenece(e, [\,]) = false$$

$$pertenece(e, x.L) = (e == x) \vee pertenece(e, L)$$

Ejercicios: Recursión estructural en listas

2. Eliminar la primera ocurrencia de un elemento de una lista.

$$\boldsymbol{elim}(\mathbf{4}, \mathbf{3}.\mathbf{4}.\mathbf{4}.\mathbf{6}.\mathbf{4}.[]) = \mathbf{3}.\mathbf{4}.\mathbf{6}.\mathbf{4}.[]$$

$$\boldsymbol{elim}: \mathbb{N} \times \textbf{Lista} \rightarrow \textbf{Lista}$$

$$\boldsymbol{elim}(\boldsymbol{e}, [\,]) = [\,]$$

$$\boldsymbol{elim}(\boldsymbol{e}, \boldsymbol{x}.\boldsymbol{L}) = \begin{cases} \boldsymbol{L} & \text{si } \boldsymbol{e} == \boldsymbol{x} \\ \boldsymbol{x}.\boldsymbol{elim}(\boldsymbol{e}, \boldsymbol{L}) & \text{en otro caso} \end{cases}$$

$$\boldsymbol{elim}(\mathbf{4}, \mathbf{3}.\mathbf{4}.\mathbf{4}.\mathbf{6}.\mathbf{4}.[]) = \mathbf{3}.\boldsymbol{elim}(\mathbf{4}, \mathbf{4}.\mathbf{4}.\mathbf{6}.\mathbf{4}.[])$$
$$= \mathbf{3}.\mathbf{4}.\mathbf{6}.\mathbf{4}.[]$$

Ejercicios: Recursión estructural en listas

3. Eliminar todas las ocurrencias de un elemento de una lista.

$$elimT(4, 3.4.4.6.4.[]) = 3.6.[]$$

$$elimT : \mathbb{N} \times Lista \rightarrow Lista$$

$$elimT(e, [\,]) = [\,]$$

$$elimT(e, x.L) = \begin{cases} elimT(e, L) & \text{si } e == x \\ x.elimT(e, L) & \text{en otro caso} \end{cases}$$

$$\begin{aligned} elimT(4, 3.4.4.6.4.[]) &= 3.elimT(4, 4.4.6.4.[]) \\ &= 3.elimT(4, 4.6.4.[]) \\ &= 3.elimT(4, 6.4.[]) \end{aligned}$$

Ejercicios: Recursión estructural en listas

4. Insertar de manera ordenada un elemento en una lista ordenada.

$$\boldsymbol{insOrd}(\mathbf{3}, \mathbf{1.2.4.}[]) = \mathbf{1.2.3.4.}[]$$

$$\boldsymbol{insOrd}: \mathbb{N} \times \text{Lista} \rightarrow \text{Lista}$$

$$\boldsymbol{insOrd}(e, [\,]) = e.[\,]$$

$$\boldsymbol{insOrd}(e, x.L) = \begin{cases} e.x.L & \text{si } e \leq x \\ x.\boldsymbol{insOrd}(e, L) & \text{en otro caso} \end{cases}$$

$$\begin{aligned} \boldsymbol{insOrd}(\mathbf{3}, \mathbf{1.2.4.}[]) &= \mathbf{1}.\boldsymbol{insOrd}(\mathbf{3}, \mathbf{2.4.}[]) \\ &= \mathbf{1.2}.\boldsymbol{insOrd}(\mathbf{3}, \mathbf{4.}[]) \\ &= \mathbf{1.2.3.4.}[] \end{aligned}$$

Ejercicios: Recursión estructural en listas

5. Ordenar una lista de menor a mayor, usando la función ***insOrd***

$$\boldsymbol{Ord}(\mathbf{3.1.4.2.}[]) = \mathbf{1.2.3.4.}[]$$

$$\boldsymbol{Ord}: \text{Lista} \rightarrow \text{Lista}$$

$$\boldsymbol{Ord}([\,]) = [\,]$$

$$\boldsymbol{Ord}(x.L) = \boldsymbol{insOrd}(x, \boldsymbol{Ord}(L))$$

Ejercicios: Recursión estructural en listas

6. Eliminar duplicados de una lista, dejando solo una ocurrencia (la primera) de cada elemento diferente

$$elimDup(3.1.2.2.1.4[]) = 3.1.2.4.[]$$

$$elimDup: \text{Lista} \rightarrow \text{Lista}$$

$$elimDup([]) = []$$

$$elimDup(x.L) = x.elimDup(elimT(x, L))$$

Ejercicios: Recursión estructural en listas

7. Determinar si una lista es palíndrome (los elementos tienen el mismo orden si se lee de izquierda a derecha que de derecha a izquierda)

$$esPalindrome(3.1.1.3.[]) = true$$

$$esPalindrome(3.1.3.[]) = true$$

$$esPalindrome(3.1.3.1.[]) = false$$

$$esPalindrome: Lista \rightarrow \{true, false\}$$

$$esPalindrome([]) = true$$

$$esPalindrome(x.[]) = true$$

$$esPalindrome(x.L.y) = (x == y) \wedge esPalindrome(L)$$