Tema 1: Recursividad

Modelo Función Recursiva No Final

```
Tipo Función f_rec(x)
Inicio
    Si   caso_base?(x) entonces
        Devolver    sol(x)
    Si_no
        Devolver    comb (f_rec(suc(x)),x)
    Fin_si
Fin_función
```

Modelo Función Recursiva Final

```
Tipo Función  f_recFinal(x,w)
Inicio
    Si    caso_base?(x) entonces
        Devolver    comb(sol(x),w)
    Si_no
        Devolver    f_recFinal(suc(x), comb(x,w))
    Fin_si
Fin_función
```

Nota:

La diferencia principal entre una función recursiva final y una recursiva no final es que mientras que en la recursiva final cada iteración es independiente de la anterior (gracias a las variables sumergidas) en la recursiva No final hay un acarreo desde la primera iteración hasta la última.

Función iterativa No final

Tipo funcion f_iter(x)

Var res,c

Inicio

C ← 0

Mientras ¬caso_base(x) **hacer**

C ← C+1

X ← suc(x)

Fin_mientras

Res ← sol(X)

Mientras C ≠ 0 **hacer**

C ← C – 1

X ← $suc(x)^{-1}$

Res ← comb(Res , X)

Fin_mientras

Devolver Res

Fin_Función

Función Iterativa Final

Tipo funcion f_iterfinal(x)

Inicio

Mientras ¬caso_base(x) **hacer**

X ← suc(x)

Fin_mientras

Devolver sol(x)

Fin_Función

Nota:

La principal diferencia entre una función recursiva final y una no final es que la No final está carente de optimización en el codigo y por tanto realiza pasos intermedios que podrían omitirse, reduciendo así el tiempo necesario para la realización

---

Elementos de una Función

(X) Representamos con la X a todos los parametros que componen la función.

Caso_base? Es la función lógica que determina si X cumple la función.

Sol(X) Calcula el valor en el caso_base?

Comb(f_rec(suc(x)),x) Combinación del valor devuelto por f_rec con uno o más parametros.

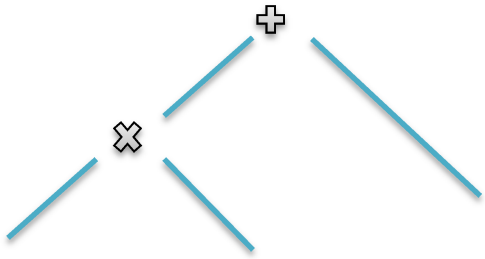
Arbol de llamadas

Cuando tenemos una función recursiva y queremos ver como se va desarrollando la misma a lo largo de las distintas iteraciones realizamos un arbol de llamadas. Este método nos permite visualizar como cada iteración va llamando a las sucesivas y a su vez estas a las siguientes.

Pasos para convertir una función recursiva No final en Final

1. **Generalización**
   Cogemos la función recursiva No final y usando como base comb(), realizamos el árbol sintáctico para conseguir las variables de inmersion necesarias para la conversión.

   Ejemplo:



2. **Sustitución y Desplegado**
   Una vez tenemos las variables de inmersión necesarias vamos sustituyendo en la función final las partes genéricas con las que hemos ido averiguando.
3. **Final**
   Una vez tenemos todos los parametros sustituidos en el modelo de la función recursiva final lo que necesitamos es averiguar los valores iniciales de los parametros de inmersión para la primera llamada de la función.

Nota:
Estos son los pasos que debe seguir uno para la realización del ejercicio, no son los descritos en los apuntes de la asignatura.

Tema 2 : Verificación Formal de Algoritmos

Base de la corrección parcial

- Anotar el comienzo y el final del programa con los asertos que ha de cumplirse.
- Anotar los puntos intermedios con asertos.
- Demostrar que cada uno de los asertos lleva al siguiente mediante las distintas reglas.

Base de la corrección Total

- Demostrar que los bucles terminan trar un número finito de iteraciones.
- Asociar a cada bucle una función monotona.

Reglas y pasos a seguir para verificar algoritmos

Lo primero que tenemos que tener claro antes de empezar con la verificación de algoritmos es que Las precondiciones se refuerzan y las postcondiciones se debilitan. Todas nuestras verificaciones se van a basar en este principio.

Un ejemplo de este principio es

X>100 → X>5

Podemos ver facilmente que el de la izquierda es mas restrictivo que el de la derecha ya que mientras que en la izquierda X tiene que ser un número que esté entre 101 e infinito en la derecha tiene que ser un número entre 6 e infinito, por tanto el rango de la derecha es mayor y eso lo hace menos restrictivo.

Partiendo de estos principios básicos llegamos a las reglas de la consecuencia:

Sí {P} S {Q} R → P
{R} S {Q} **Primera regla de la consecuencia, Reforzar la precondición.**

Sí {P} S {Q} Q → R
{P} S {R} **Segunda regla de la consecuencia, Debilitar la postcondición.**

La asignación

Para demostrar una verificación utilizamos la regla de la asignación. Esta podemos aplicarla directamente sin tener que explicar nada más ya que su verificación es inmediata. Su verificación se hace de “abajo a arriba”. Este concepto se comprende mejor con un ejemplo por delante:

{A ≠ 0 ∧ n>0 }
X ← 1
{A ≠ 0 ∧ n>0 ∧ x=1 }

Para verifitacar esta especificación partimos desde la postcondición (desde abajo):

{A ≠ 0 ∧ n>0 ∧ 1=1 }
X ← 1
{A ≠ 0 ∧ n>0 ∧ x=1 }

Partiendo de la postcondición llegamos a una nueva precondición. Ahora es el momento de recordar la frase:

Las precondiciones se refuerzan y las postcondiciones se debilitan

Por tanto para que la precondición que nos dieron sea correcta se tiene que cumplir que:

A ≠ 0 ∧ n>0 → A ≠ 0 ∧ n>0 ∧ 1=1

Se ve directamente que se cumple ya que 1=1 es una tautología.

Regla de composición

Mediante la regla de la composición podemos demostrar más de una instrucción cuando vengan de forma consecutiva. Diremos que la especificación

{P}
S_1
S_2
{Q}

Es correcta si existe un aserto R tal que se cumplan las condiciones de verificación generadas por:

{P} S_1 {R}
{R} S_2 {Q}

Es decir, si se cumplen los asertos intermedios entre dos o más instrucciones, apoyandonos en la regla de la composición podemos decir que es correcto.

Reglas del Condicional

Para verificar la correccion de una estructura selectiva simple, diremos que la especificación

{P}

Si B **entonces**

S

Fin_si

{Q}

Es correcta si satisface las condiciones de verificación:

- P ∧¬B → Q
- Y las generadas por {P ∧ B} S {Q}

Nota: La B de estas condiciones es la "B" de la estructura , que en nuestro caso, lo normal es que sea el caso_base?

Regla del Mientras

Para poder realizar la verificacion de un bucle mientras necesitamos un predicado que recible el nombre de invariante el cual describe los distintos estados por los que pasa el bucle. Diremos que la especificación

{P}

Mientras B **hacer** {I}

S

Fin_mientras

{Q}

es correcta, donde I denota el conjunto de condiciones del invariante que deben cumplirse en todo momento. Para ello deben satisfacerse las condiciones de verificación generadas por:

- P → I
- I ∧ ¬B → Q
- Y las generadas por {I ∧ B} S {I}

Estas son las necesarias para una verificación parcial, para una verificación total hay que añadir:

- I ∧ B → t>0
- Y las generadas por {I ∧ B ∧ t=T} S {t<T}

Donde t es una función monotona que varía con cada iteración del bucle y nos sirve para demostrar que este sólo realiza un número finito de iteraciones. Esta función recibe el nombre de función de cota o función limitadora.

Nota: Para calcular correctamente el invariante lo mejor es realizar un ejemplo e ir viendo paso a paso los valores que toma cada variable y como va evolucionando en cada iteración.

Verificacion de Algoritmos Recursivos

En todo algoritmos recursivo aparece al menos una estructura selectiva doble en la que una de las ramas comprueba el caso_base? y la otra corresponde al caso general, en la cual se genera la recursividad.

Para demostrar la correccion de un algoritmo recursivo se tiene que cumplir que:

- P → B v ¬B
- P ∧ B → Q
- P ∧ ¬B → P
- P ∧ ¬B ∧ Q → Q

Nota: Demostrando estos puntos conseguimos demostrar que es parcialmente correcto que en teoría es lo máximo que nos van a pedir en el examen.

Tema 3 : Diseño Modular

Ciclo de Vida de un Software

- **Definición:** Qué hace el programa
- **Desarrollo:** Cómo lo hace, es decir, el código del mismo.
- **Mantenimiento:** Cambios realizados ya sea por mejora del código o para corregir bugs del mismo.

Podemos decir que esta es la visión clásica y sencilla ya que si quisieramos dar un ciclo de vida más detallado nos pararíamos en 5 puntos principales:

1. **Analisis:** Estudio y especificación de los requisitos que debe cumplir el programa.
2. **Diseño:** El como lo va a hacer.
3. **Implementación:** Se codifica.
4. **Pruebas:** Se buscan errores en el codigo.
5. **Mantenimiento:** Mejoras o correcciones.

Módulo

Definimos un modulo como la unidad de organizacion de un algoritmo que realiza funciones específicas con un objetivo diferenciado del resto de módulos. Consta de elementos públicos y privados. Cada módulo tiene que estar bien documentado para que se sepa que hace en cada momento al mirar su código.

Criterios de Descomposicion:

- **Por Tareas:** Un módulo por operación.
- **Por datos:** Según los tipos de datos y las operaciones relacionadas con los mismos.

Relación entre módulos:

- **Cohesión:** Todo aquello que esté relacionado en el código debe intentar estar ubicado en el mismo módulo.
- **Acoplamiento:** Los módulos tienes que intentar ser lo más independiente posible, es decir, intentar que no importen mucha información de otros módulos.

Esquema de un módulo:

Modulo Nombre

Importa

Relación de módulos de los que se importa algo

Fin_Importa

Exporta

Relación de elementos exportables.

Fin_Exporta

Implementación

Declaraciones, Definiciones, Procedimientos y Funciones.

Fin_implementación

Fin_Modulo

Nota: Como usuario sólo tenemos acceso a los elementos exportables de un módulo. Como administrador del mismo hay que decidir que elementos de un módulo van a ser públicos y cuales van a ser privados.

En C un módulo se compone de un archivo .h con la importacion y la exportación y un .C con la implementación.

Transformación de un módulo en Pseudocódigo a uno en C

1. Creamos Módulo.h con todoas las declaraciones de los elementos públicos o exportables (sección de exportación). Cabe recordar que hay que incluir todas las constantes o tipos que se quieran hacer públicos.
2. Creamos Módulo.c con la definicion o implementación de todas las funciones, tanto públicas como privadas. Para que una función sea privada es necesario que a la hora de declararla vaya precedida de la palabra “static”. Dentro del archivo Módulo.c hay que hacer un #include al archivo Módulo.h

PseudoCódigo	C
Importación	#Include
Exportación	.h
Implementación	.c

Tema 4: Analisis de Algoritmos

Para que una computadora sea capaz de resolver un problema es necesario descomponerlo en operaciones sencilla. Estas operaciones sencillas deben de cumplar una serie de requisitos. Deben de estar bien definidas, es decir, que no quepa duda de lo que realiza dicha operación y deben estar acotadas tanto en espacio como en tiempo (nada de bucles infinitos).

Se suele creer que no hace falta buscar la eficiencia en los algoritmos por el hecho de que los ordenadores son cada vez mas pontentes y cada vez pueden resolver los mismos algoritmos en menos tiempo. El problema es que mientras que la capacidad de mejora de los ordenadores sigue una función geometrica (es decir, lo normal es que cada X años los ordenadores dupliquen su capacidad), el tiempo de resolución de un algoritmo no crece de forma geometrica, sino mucho más rapido. Por tanto, por mucho que mejoren los ordenadores, sino buscamos la eficiencia en el código, habrá problemas cuya resolución este fuera de nuestro alcance.

A la hora de implementar un Algoritmo en dos computadoras distintas sabemos que como mucho si en uno la velocidad de ejecución es de X segundos en el otro como mucho tardará cX segundos, donde c es una constante que multiplica a X.

A la hora de estudiar el tiempo que tarda en ejecutarse un algoritmo no nos sirve calcular cuanto tardará en el caso mas favorable, ya que se entiende que dicho caso se dará en pocas ocasiones y nos daría una estimación muy a la baja del tiempo que necesita. Por otro lado, tampoco es del todo correcto hacer la estimación media, ya que esta está muy ajustada y tampoco sabemos si el promedio es que el 50% de casos sean casos favorables y el otro 5 en el peor de los casos. Por ello las empresas, a la hora de estudiar sus algoritmos lo normal es que los estudien mediante el método más desfavorable, de esa forma se aseguran que pase lo que pase, el algorimo, como mucho tardará ese tiempo.

Falta el calculo de orden de los algoritmos, lo añadiré entre hoy y mañana!

Un Saludo, Paco Maestre