GUÍA DE APRENDIZAJE PARA CONCURSANTES ICPC Y IOI: RECURSIVIDAD

1. Introducción

La recursividad es una técnica de programación que se utiliza para realizar una llamada a una función desde ella misma, de allí su nombre.

Un **algoritmo recursivo** es un algoritmo que expresa la solución de un problema en términos de una llamada a sí mismo. La llamada a sí mismo se conoce como llamada recursiva o recurrente.

Existe un grupo de problemas que su solución radica en la implementación de algoritmos que usen esta técnica e incluso existen casos donde la utilización de esta técnica es mucho más óptimo de que una solución iterativa.

2. Conocimientos previos

2.1. Concepto de función o método

Las aplicaciones informáticas que habitualmente se utilizan, incluso a nivel de informática personal, suelen contener decenas y aún cientos de miles de líneas de código fuente. A medida que los programas se van desarrollando y aumentan de tamaño, se convertirían rápidamente en sistemas poco manejables si no fuera por la modularización, que es el proceso consistente en dividir un programa muy grande en una serie de módulos mucho más pequeños y manejables. A estos módulos se les ha solido denominar de distintas formas (subprogramas, subrutinas, procedimientos, funciones, métodos etc.) según los distintos lenguajes. Sea cual sea la nomenclatura, la idea es sin embargo siempre la misma: dividir un programa grande en un conjunto de subprogramas o funciones más pequeñas que son llamadas por el programa principal; éstas a su vez llaman a otras funciones más específicas y así sucesivamente.

La división de un programa en unidades más pequeñas o funciones presenta –entre otras– las ventajas siguientes:

1. **Modularización** Cada función tiene una misión muy concreta, de modo que nunca tiene un número de líneas excesivo y siempre se mantiene dentro de un tamaño manejable. Además, una misma función (por ejemplo, un producto de matrices, una resolución de un sistema de ecuaciones lineales, ...) puede ser llamada muchas veces en un mismo programa, e incluso puede ser reutilizada por otros programas. Cada función puede ser desarrollada y comprobada por separado.

2. **Ahorro de memoria y tiempo de desarrollo** En la medida en que una misma función es utilizada muchas veces, el número total de líneas de código del programa disminuye, y también lo hace la probabilidad de introducir errores en el programa.

3. **Independencia de datos y ocultamiento de información** Una de las fuentes más comunes de errores en los programas de computador son los efectos colaterales o perturbaciones que se pueden producir entre distintas partes del programa. Es muy frecuente que al hacer una modificación para añadir una funcionalidad o corregir un error, se introduzcan nuevos errores en partes del programa que antes funcionaban correctamente. Una función es capaz de mantener una gran independencia con el resto del programa, manteniendo sus propios

datos y definiendo muy claramente la interfaz o comunicación con la función que la ha llamado y con las funciones a las que llama, y no teniendo ninguna posibilidad de acceso a la información que no le compete.

Una función de C++ o Java es una porción de código o programa que realiza una determinada tarea. Una función está asociada con un **identificador o nombre**, que se utiliza para referirse a ella desde el resto del programa. En toda función utilizada en C++ o Java hay que distinguir entre su **definición**, su **declaración** su **llamada**, su **valor de retorno** y sus **argumentos**. En algunos casos tanto la **definición** y su **declaración** se realizan en conjunto.

2.2. La pila de recursión

La memoria de un ordenador se divide en 4 segmentos:

- Segmento de código: almacena las instrucciones del programa en código máquina
- Segmento de datos: almacena las variables estáticas o constantes.
- Montículo: almacena las variables dinámicas
- Pila del programa: Parte destinada a las variables locales y parámetros de la función que se está ejecutando.

3. Desarrollo

Lo primero que debemos entender las diferencias de como funciona un metodo lineal o tradicional a uno recursivo.

Cuando llamamos a una función (o método) el proceso es el siguiente:

1. Se reserva espacio en la pila para los parámetros de la función y sus variables locales.
2. Se guarda en la pila la dirección de la línea del código desde donde se ha llamado al método.
3. Se almacenan los parámetros de la función y sus valores en la pila.
4. Finalmente se libera la memora asignada en la pila cuando la función termina y se vuelve a la llamada de código original.

En cambio, cuando la función es recursiva:

- Cada llamada genera una nueva llamada a una función con los correspondientes objetos locales
- Volviéndose a ejecutar completamente, hasta la llamada a si misma. Donde vuelve a crear en la pila los nuevos parámetros y variables locales. Tantos como llamadas recursivas generemos.
- Al terminar, se van liberando la memoria en la pila, empezando desde la última función creada hasta la primera, la cual será la ultima en liberarse.

En la implementación de un algoritmo recursivo consta de dos partes:

- Caso base: Es la resolución del problema de manera directa
- Caso Recursivo: Caso en el que el problema se divide en versiones más pequeñas de si mismo.

Es importante hacer notar que en la implementación de los algoritmos recursivos puede existir más de un caso base y más de un caso recursivo lo que siempre debe existir uno de cada tipo porque la ausencia de uno de estos casos significa que o tenemos un algoritmo no recursivo (si falta el caso recursivo) o un algortimo que provoca un bucle sin fin (si falta el caso base)

3.1. Cómo diseñar un algoritmo recursivo

1. Reconocer el caso base y proporcionar una solución para él.
2. Diseñar una estrategia para dividir el problema en versiones más pequeñas del mismo considerando avanzar hacia el caso base.
3. Combine las soluciones de los problemas más pequeños para obtener la solución del problema original.

Según el modo en que se realiza la llamada recursiva se puede clasificar o agrupar como:

- **Directa:** Cuando un método/función se invoca así mismo.
- **Indirecta o mutua :** Cuando un método/función puede invocar a una segunda función/método que a su vez invoca a la primera
- **Recursión lineal no final:** En la recursión lineal no final el resultado de la llamada recursiva se combina en una expresión para dar lugar al resultado de la función que llama.
- **Recursión lineal final:** En la recursión lineal final el resultado que es devuelto es el resultado de ejecución de la última llamada recursiva.
- **Recursión múltiple:** Alguna llamada recursiva puede generar más de una llamada a la función.

4. Implementación

Como lo abordado no es un algoritmo sino una técnica de progrmación veamos algunos ejemplos de la implementación de esta técnica para resolver determinados problemas.

```
/* Ejemplo de recursividad de indirecta o mutua para determinar si un numero
   positivo es
par o impar */

int par(int n){
  if (n==0) return 1;
```

```
  else return (impar(n-1));
}
int impar(int n){
  if (n==0) return 0;
  else return(par(n-1));
}

/* Calcular el ensismo termino de Fibanacci*/
long fibonacci(int n)
{
  if(1 == n || 2 == n) {
    return 1;
  } else {
    return (fibonacci(n-1) + fibonacci(n-2));
  }
}

/*Calcular el maximo comun divisor*/
long mcd(long a, long b){
  if (a==b) return a;
  else if (a<b) return mcd(a,b-a);
  else return mcd(a-b,b);
}

/*Calcular el factorial*/
int factorial(int numero){
   if (numero > 1) return numero*factorial(numero-1);
   else return 1;
}
```

5. Complejidad

La complejidad de algoritmos recursivos involucra la solución de una ecuación diferencial. El método mas simple es adivinar una solución y verificar si esta bien la adivinanza.

Recurrencia matemática: Define una función en f términos de ella misma. En el ejemplo del factorial de un numero natural, tenemos: $n! = n(n-1)$.

En el ejemplo de los números Fibonacci

$F_n = F_{n-1} + F_{n-2}; n >= 2$

$F_0 = 0$

$F_1 = 1$

(Donde n, 1, 0 son las bases de F).

Algoritmo factorial recursivo

Dado n Factorial n*F(n-1) devolver factorial

Procedimiento F(k) Si k=0 o 1 entonces devolver 1 caso contrario devolver k*F(k-1)

La complejidad de un algoritmo recursivo se expresa a través de una ecuación de recurrencia:

Sea T(n) = tiempo para calcular n!, entonces:

$T(n) = T(n-1) + 1; sin >= 1$

$T(1) = 1$

Si resolvemos de una manera mas sencilla esta ecuación tenemos:

$T(n) = T(n-1) + 1$

$T(n) = (T(n-2) + 1) + 1 = T(n-2) + 2$

$T(n) = ((T(n-3) + 1) + 1) + 1 = T(n-3) + 3$

...

$T(n) = T(n-k) + k$

...

$T(n) = T(1) + n - 1$

$T(n) = T(0) + n = n$

Luego la complejidad del algoritmo factorial recursivo es $O(n)$.

5.1. Reducción por sustracción

Si el tamaño n del problema decrece en una cantidad constante b en cada llamada, se realizan a llamadas recursivas y las operaciones correspondientes a la parte no recursiva del algoritmo toman un tiempo $O(n^k)$ entonces:

$T(n) = aT(n-b) + O(n^k); si \quad n >= b$

La solución de esta ecuación de recurrencia es de la forma:

$$T(n) = \begin{cases} n^k, & si \quad a < 1 \\ n^k + 1, & si \quad a = 1 \\ a^n/b, & si \quad a > 1 \end{cases}$$

5.2. Reducción por división

Si el algoritmo con un problema de tamaño n realiza a llamadas recursivas con subproblemas de tamaño n/b y las operaciones correspondientes a la parte no recursiva, que corresponde a descomponer el problema en los subproblemas y luego combinarlas soluciones de estos, toman un tiempo O(n^k) entonces:

$T(n) = aT(n/b) + O(n^k); si \quad n >= b$

La solución de esta ecuación de recurrencia es de la forma:

$$T(n) = \begin{cases} n^k, & si \quad a < b^k \\ n^k \log n, & si \quad a = b^k \\ n^{\log_b a}, & si \quad a > b^k \end{cases}$$

5.3. Complejidad para los algoritmos recursivos del tipo divide y venceras

Estos algoritmos recursivos tienen asociada una ecuación de recurrencia tipo, cuya resolución es lo que se conoce como el *Teorema Maestro*.

A continuación se explica cómo interpretar dicho teorema maestro.

Aplicar el teorema maestro implica identificar diversos parámetros de un algoritmo recursivo tipo divide y vencerás. Dichos parámetros son:

- **a**: Número de llamadas recursivas que se realizan en el caso recursivo
- **b**: Factor por el cual se divide el tamaño del problema en cada llamada recursiva
- **k**: Mayor exponente de las polinomios asociados a las complejidades del caso base y de la parte no recursiva de la rama recursiva del algoritmo, asumiendo que en ambos casos se trata de complejidades polinómicas

Por tanto, el procedimiento para aplicar el teorema maestro es tal como sigue:

1. Identificar cuáles son las ramas recursivas del algoritmo y cuáles corresponden al caso base.
2. Identificar el número de llamadas recursivas (parámetro a) que se realizan en el caso recursivo.
3. Identificar la razón (parámetro b) por la cual se divide el tamaño del problema en cada llamada recursiva.
4. Calcular la complejidad del caso base, como si se tratase un algoritmo iterativo normal. La complejidad resultante, para aplicar el teorema maestro, debe ser polinómica. Sea k el grado de dicho polinomio. Si el algoritmo tiene complejidad constante, es decir, O(1), k=0, ya que $n^0 = 1$.
5. El teorema maestro asume que las complejidades del caso base y de la parte iterativa del caso recursivo son polinómicas y poseen el mismo grado. Por tanto, tomaremos como dicho grado al mayor de los grados de los polinomios asociados a las complejidades del caso base y de la parte iterativa del caso recursivo. Es decir, $k = max(casobase^k, casorecurrente^k)$. Esto es posible gracias a que un algoritmo de complejidad O(n^a) se puede siempre convertir en otro equivalente de complejidad O(n^b), siempre que $b > a$. Pensad que bastaría con anidar el algoritmo de complejidad O(n^a) en tantos bucles sin sentido como hiciese falta para incrementarle la complejidad hasta que llegase a b.
6. A continuación se calcula la relación entre a y b^k y se aplica el teorema maestro, sustituyendo los valores de k y a tal como corresponda.

6. Aplicaciones

Podemos utilizar recursividad para reemplazar cualquier tipo de bucle. A pesar de ello en la práctica no se utiliza demasiado, debido a que un error puede ser trágico en la memoria, así como tener una lista con millones de datos, puede hacer que utiliza mucha memoria. Aun así, la gran mayoría de las veces, utilizamos recursividad para algoritmos de búsqueda u ordenación.

La razón fundamental es que existen numerosos problemas complejos que poseen naturaleza recursiva, por lo cual son más fáciles de implementar con este tipo de algoritmos. Sin embargo, en condiciones críticas de tiempo y de memoria, la solución a elegir debe ser normalmente de forma iterativa siempre que sea posible.

7. Ejercicios propuestos

A continuación una lista de ejercicios que se resuelven utilizando esta técnica:

- DMOJ - Aplicando Reverso
- DMOJ - El problema de Josephus
- DMOJ - Generating Words
- SPOJ - DYZIO - Dyzio