**APUNTES MASTERING DATA STRUCTURES & ALGORITHMS**

**Tipos de funciones recursivas**

**Tail Recursion.-** Cuando la llamada a la propia función se hace al final del código de la función. Es decir no se ejecutan tareas en el “regreso”. Estas funciones son fáciles de convertir en bucles ya que la estructura es similar, y la ventaja de estos últimos es que, aunque a nivel de tiempo ambas son O(n), a nivel de espacio la función recursiva es O(n), mientras que el bucle es O(1) y por ende mas eficiente. (Muchos compiladores aprovechan este hecho para convertir Tail Recursions en loops).

**Head Recursion.-** La recursión se da al principio del código. Se ejecutan mas tareas en el retorno, mientras que no se ejecuta nada en el “ascenso”. Es mas difícil de convertir en bucle, ya que la condición base cambia. Complejidad de tiempo y espacio es similar a Tail Recursion.

**Tree Recursion.-** Se distingue de la recursión lineal, que solo se llama a si misma una vez, por llamarse multiples veces en una función. La complejidad de tiempo es geométrica O(x^n) donde x es el numero de veces que se llama dentro de la funcion. La complejidad de espacio es igual al “alto” del árbol = O(n).

**Indirect Recursion.-** Cuando una función llama a otra que a la vez llama a la primera.

**Suma de números naturales con recursión.-** Formula incluye recursión:

/ 0 n = 0 O(n) calls: n + 1

Sum(n)

\ Sum(n-1)+n n > 0

Las definiciones matemáticas con recursión son fácilmente convertibles a funciones recursivas.

Se puede lograr una formula para evitar la recursión. En este caso : n (n+1) / 2

Esto hará que la complejidad de tiempo sea O(1).

**Factorial con recursión.-**

/ 1 n = 0 O(n) calls: n + 1

Fact(n)

\ fact(n-1) \* n n > 0

**Exponential con recursion.-** m multiplied n times:

/ 1 n = 0 O(n) calls: n+1

Pow(m, n)

\ pow (m, n-1) \* m n > 0

El numero de llamadas puede ser reducido a la mitad si es par agrupando llamadas. Ej m=2 n =8:

2^8 = (2^2)^4. Si es impar es la mitad + 1

**Taylor series (e^x).-**  Se combina los tres anteriores: suma, factorial y exponencial. Debe usar multiples valores pero una sola suma, por lo que se utiliza variables estaticas.

e^x = 1 + x/1 + x^2/2! + x^3/3! + ….

Función e(x,n) = r + p / f

Variable r almacena suma. Variables estaticas p y f, inicializadas a 1. La p se encargara del exponencial, al multiplicarse por x en cada recursión, y f de factorial al multiplicarse por n en cada recursión. (Se hace en el retorno)

Complejidad = O (n^2). El numero de multiplicaciones es n (n+1) cuadrática.

Formula = 1 + x/1 + x^2/2! + x^3/3! + x^4/4!

Multiplicaciones= 0 0 2 4 6 total = 12

Se puede reducir a O(n) si se multiplica números comunes (se hace lineal). Ej:

Formula = 1 + x / 1 ( 1 + x / 2 ( 1 + x / 3 ( 1 + x / 4 ) ) )

Multiplicaciones = 0 1 1 1 total = 3

**Fibonacci.-**

/ 0 n = 0

Fib(n) 1 n = 1

\ fib(n-2) + fib(n-1) n > 1

Complejidad = O(2^n); recursión tree. Es una recursión excesiva, porque se llama múltiples veces con el mismo parámetro.

Se puede reducir el numero de llamadas mediante “memorización”: Se guarda los valores de las llamadas conocidas en un array, y si ya se conoce el valor, no se hace una llamada, sino que solo se utiliza el valor guardado.

**Combination Formula.-**

nCr = n!/r!(n-r)!

se utiliza el triangulo de Pascal:

1(0C0)

1(1C0) 1(1C1)

1(2C0) 2(2C1) 1(2C2)

1(3C0) 3(3C1) 3(3C2) 1(3C3)

1(4C0) 4(4C1) 6(4C2) 4(4C3) 1(4C4)

Cada valor se obtiene sumando (n-1Cr-1) + (n-1Cr). Ej. 4C2 = 3C1 + 3C2

**Tower of Hanoi.-** complejidad exponencial O(2^n)

Programa básico:

void TOH(int n, int A, int B, int C) {

    count++;

    if (n>0) {

        TOH(n-1, A, C, B);

        movement++;

        std::cout << movement << ". Move disc " << n << " from column " << A << " to column " << C << std::endl;

        TOH(n-1, B, A, C);

    }

}

Entonces, el orden de argumentos es (numero de discos, columna A, columna B, columna C)

Detro de la función: primero recursivo en (n-1, A, C, B), imprimir movimiento de A a C, y luego recursivo en (n-1, B, A, C). El movimiento se realiza al retornar de la primera recursión.

ARRAYS

2D arrays

En la memoria el almacenaje es 1D, linear. Si se inicializa un array[3][4] = 3 \* 4 = 12 direcciones en memoria. Array [m][n]

Formas de representación:

**Row Major Mapping**.- Los elementos se almacenan fila a fila. En el ejemplo anterior de [3][4], los elementos de las columnas se almacenarían seguidamente, 4 por las 3 columnas.:

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| A00 | A01 | A02 | A03 | A10 | A11 | A12 | A13 | A20 | A21 | A22 | A23 |

La conversión para obtener la dirección es:

Add(A[1][2]) = 200 + (1 \* 4 + 2) \* 2 = 212

Donde: Add es la dirección. Se asume una dirección de 200. 1 es el numero de índice. 4 es el numero de columnas por fila. 2 es el numero de bytes del int (se asume como 2. Varia de acuerdo al compilador). El resultado de la memoria es 212.

Add(A[2][3]) = 200 + (2 \* 4 + 3) \* 2 = 222

Entonces la formula es = (A[i][j]) = Lo + ( i \* n + j ) \* w. Va de izquierda a derecha

Donde Lo es la dirección del primer elemento, i \* n es la fila, j es la columna, y w es el espacio en bytes que ocupa el elemento.

**Column Major Mapping.-**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| A00 | A10 | A20 | A01 | A11 | A21 | A02 | A12 | A22 | A03 | A13 | A23 |

Formula:

([i][j]) = Lo + ( j \* m + i ) \* w. Va de derecha a izquierda

**Formula para arrays de nDs.-**

Ejemplo 4D:

Tipo A[d1][d2][d3][d4]

Row-major, de izquierda a derecha:

Add(A[i1][i2][i3][i4]) = Lo + (i1 \* d2 \* d3 \*d4 + i2 \* d3 \*d4 + i3 \* d4 + i4) \* w.

\*d = dimension, i= índice.

Column major, de derecho a izquierda:

Add(A[i1][i2][i3][i4]) = Lo + (i4 \* d1 \* d2 \* d3 + i3 \* d1 \* d2 – i2 \* d1 – i1) \* w.

Complejidad O(n^2) si se hace todas las multiplicaciones. Con elementos comunes se puede reducir las multiplicaciones a O(n). (Esta es la ley de Horner)

ADT arrays

**Agregar** tiene una complejidad de O(1) (constante), porque tiene dos pasos (siguiendo modelo de estructura de Array{ A, size, length):

A[length] = x; 1

Length++; 1

2 operaciones constantes.

**Insertar,** parte de la anterior operación. Despues de añadir un elemento se mueven los elementos hacia la derecha, desde el final hasta el índice objetivo.

Ej. Insert(índex = 4, element=15)

For (i=length; i > index ; i--) {

A[ i ] = A[ i-1 ]; 0 – n (depende del indice)

}

A [ index ] = element; 1

Length++; 1

La complejidad de tiempo es: min(1), max(n) = O(n).

**Eliminar,** una vez eliminado el elemento, no se debe dejar el espacio vacio, asi que se debe mover todos los elementos posteriores hacia la izquierda. (el movimiento es hacia la izquierda hasta length -1):

X = A[index]; 1

For (int i = index; i < length; i++) {

A[ i ] = A [ i + 1]; 0-n

Length--; 1

Complejidad de tiempo: min(2), max = n + 2 = O(n).

**Busqueda lineal**. Mediante un loop se compara elemento buscado índice a índice. Complejidad de tiempo mínimo = O (1), máximo = O(n); Si la búsqueda no tiene éxito O(n); AVG = (1+2+3+… +n)/n = (n+1)/2 = O(n).

\*Usualmente el peor caso se corresponde con el caso promedio

Metodos para mejorar eficiencia de búsqueda lineal:

**Transposición.** Se puede mejorar el tiempo de la búsqueda lineal moviendo el numero encontrado un lugar hacia la izquierda. De esta manera, los elementos mas buscados se encontraran al principio y tardaran menos la siguiente vez que se los busque.

**Mover al frente .-** (move to head). El cambio se hace con el primer elemento, en lugar de con el anterior.

**Busqueda binaria.** Solo sirve en listas ordenadas. Se va dividiendo la lista por la mitad y se verifica si el valor buscado es mayor o menor a esa mitad, modificando los limites de forma progresiva.