Programación Avanzada

Pontificia Universidad
JAVERIANA
Colombia

Ing. Orlando Vasquez

Recursividad

Como ya sabemos, un subprograma puede llamar a cualquier otro subprograma y éste a otro, y así sucesivamente; dicho de otro modo, los subprogramas se pueden anidar.

Cuando una función se llama a sí misma dentro de su bloque de código, decimos que la función se auto-llama ó auto-invoca. Este proceso de auto-llamado de una función se denomina recursividad.

La recursividad se usa para crear un comportamiento similar a un bucle usando una función y sin usar declaraciones de bucle.

La escritura de un procedimiento o función recursiva es similar a sus homónimos no recursivos; sin embargo, para evitar que la recursión continúe indefinidamente **es preciso incluir una condición de terminación**.

Llamado	Retorno

function a(){
  b();
}
function b(){
  c();
}
function c(){
  ……
  ……
}
	c-Retorna a **b**
	b- retorna a **a**

Llamado	Retorno

function a(){
  a();
}
	a-Retorna a **a**

- Luis, J. A., & Aguilar, L. J. (2008). *Fundamentos de programación*. McGraw-Hill Education.
- Castillo Reyes, O. Análisis de Algoritmos - Recursividad. https://www.uv.mx/personal/ocastillo/files/2011/04/Recursividad.pdf

Recursividad

Porque usar recursividad

- Son mas cercanos a la descripción matemática.
- Generalmente mas fáciles de analizar
- Se adaptan mejor a las estructuras de datos recursivas.
- Los algoritmos recursivos ofrecen soluciones estructuradas.

Cuando/Porque utilizar recursividad

- Para simplificar el código.
- Cuando la estructura de datos es recursiva ejemplo : árboles.
- Es la forma mas semejante a la formula matemática real

Castillo Reyes, O. Análisis de Algoritmos - Recursividad. https://www.uv.mx/personal/ocastillo/files/2011/04/Recursividad.pdf

Recursividad

Caso 1

Muchas funciones matemáticas se definen recursivamente. Un ejemplo de ello es el factorial de un número entero n.

La función factorial se define como :

n! = 1 si n es igual a cero

En caso contrario

n! = n * (n-1)*(n-2)*(n-3)*….2*1

Es decir

5!=5*4*3*2*1

4!=4*3*2*1

3!=3*2*1

2!=2*1

1!=1

Lo cual indica que:

5!=5*4!

4!=4*3!

3!=3*2!

2!=2*1!

1!=1*0!

```
#include <iostream>
using namespace std;
int factorial (int n);
int main(){
   int n;
   int fct=0;
   cout<< "Digite el numero para calcular factorial:";
   cin >>n;
   fct=factorial(n);
   cout<< n<<"!="<<fct;
   return 0;
}

int factorial (int n){
     int f=0;
     if (n==0)
          f=1;
     else{
           f=n*factorial(n-1);
           cout<< n<<"!="<<f<<endl;
      }
      return f;
}
```

factorial.cpp

Pontificia Universidad JAVERIANA Colombia

Recursividad

Caso 2

Convertir decimal a binario:

Para convertir un decimal a binario, se deben realizar divisiones sucesivas entre 2, almacenando el residuo en cada iteracion y tomando como dividendo el cociente para la proxima iteracion

Convertir decimal a binario		
Decimal	Divisor	Residuo
23	2	1
11	2	1
5	2	1
2	2	0
1	2	1
0	2	0
Resultado	10111	

Decimal	Divisor	Residuo
18	2	0
9	2	1
4	2	0
2	2	0
1	2	1
0	2	0
Resultado	10010	

```
#include<iostream>
#include<cstdlib>
using namespace std;
int binario(int n){
if(n>0)
  binario(n/2);
   cout<<n%2;
}
int main(){
    int nro;
    cout<<"Convertir a binario un numero decimal "<<endl;
    cout<<" Ingrese el numero: ";
    cin>>nro;
    cout<<"\nNumero:"<<nro<<endl;
    cout<<"Binario:";
    binario(nro);
    return 0;
}
```

binario.cpp

Pontificia Universidad JAVERIANA Colombia

Divide y Venceras

La descomposición de un programa en módulos independientes más simples se conoce también como el método de divide y vencerás (divide and conquer). Cada módulo se diseña con independencia de los demás, y siguiendo un método ascendente o descendente se llegará hasta la descomposición final del problema en módulos en forma jerárquica.

Los problemas complejos se pueden resolver más eficientemente cuando se rompen en subproblemas que sean más fáciles de solucionar que el original.
Es el método de divide y vencerás mencionado anteriormente, y que consiste en dividir un problema complejo en otros más simples.
Ej: El problema de encontrar la superficie y la longitud de un círculo se puede dividir en tres problemas más simples o subproblemas
1) leer datos de entrada; 2) calcular superficie y longitud de circunferencia, y 3) escribir resultados

La descomposición del problema original en subproblemas más simples y a continuación la división de estos subproblemas en otros más simples que pueden ser implementados para su solución se denomina diseño descendente (top-down design). Normalmente, los pasos diseñados en el primer esbozo del algoritmo son incompletos e indicarán sólo unos pocos pasos. Tras esta primera descripción, éstos se amplían en una descripción más detallada con más pasos específicos. Este proceso se denomina refinamiento

Luis, J. A., & Aguilar, L. J. (2008). *Fundamentos de programación*. McGraw-Hill Education.

Recursividad - Divide y Venceras

Caso 1

Búsqueda binaria: Es un algoritmo de búsqueda que encuentra la posición de un valor en un arreglo ordenado. Compara el valor con el elemento en el medio del arreglo, si no son iguales, la mitad en la cual el valor no puede estar es eliminada y la búsqueda continúa en la mitad restante hasta que el valor se encuentre. Sino se encuentra, devuelve menos -1

Posicion	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
Datos	2	4	6	8	11	13	15	19	23	34	43	52	60	62	63	67	71	79	81
Buscar	43																		

Iteracion 1	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	
	2	4	6	8	11	13	15	19	23	34	43	52	60	62	63	67	71	79	81	Mitad 9

Iteracion 2	9	10	11	12	13	14	15	16	17	18	Mitad 13
	34	43	52	60	62	63	67	71	79	81	

Iteracion 3	9	10	11	12	Mitad 10
	34	43	52	60	

Iteracion 4	10	11	12	Mitad 10
	43	52	60	

Iteracion 5	Inicio=Fin=10
Resultado	posicion = 10

busquedaBinaria.cpp

Pontificia Universidad JAVERIANA Colombia

Complejidad del algortimo

Búsqueda Binaria vs Búsqueda lineal

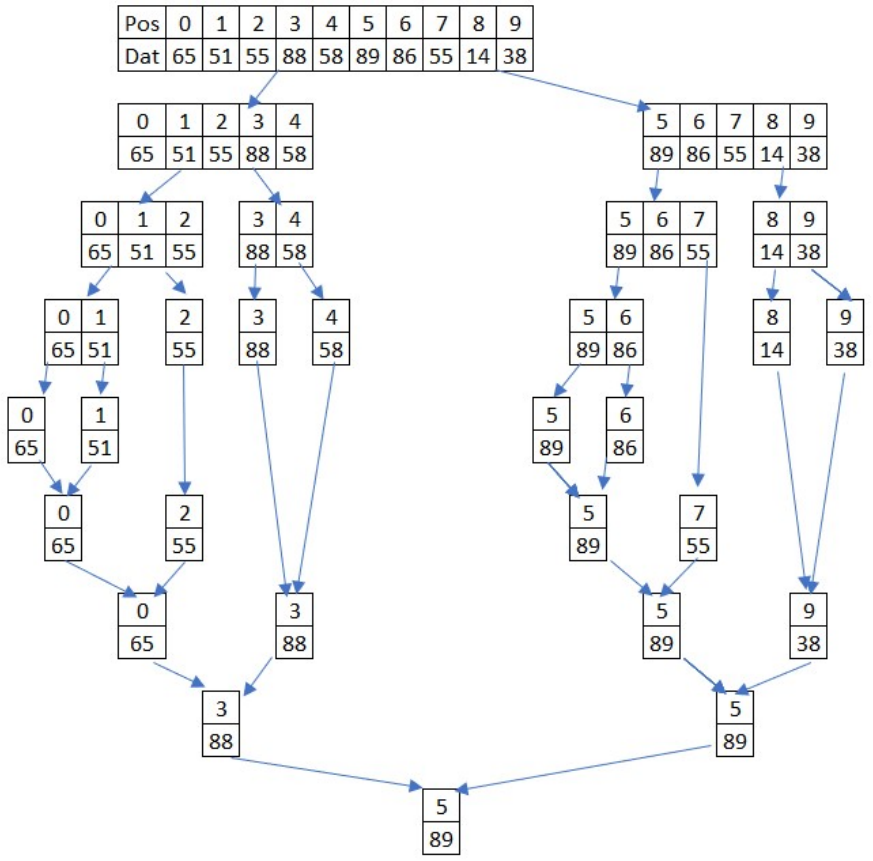
La complejidad del algoritmo de busqueda binaria esta dada por la formula **Round (Log $_2$n** +1**)** , donde **n** es el numero de posiciones del arreglo. Es decir en un arreglo de 16 posiciones, el número máximo de iteraciones posibles para encontrar un numero es Log $_2$ 16+1=5, si el arreglo tiene 32 posiciones, el número máximo de iteraciones será 6 porque Log $_2$ 32+1=6

Si lo comparamos con el algoritmo de búsqueda lineal, este tiene una complejidad (**n)**, esto quiere decir que el número máximo de iteraciones esta dado por el numero de posiciones del arreglo.

Es decir si el arreglo es de 16 posiciones, el número máximo de iteraciones es 16, si el arreglo es de 32, el número máximo de iteraciones será 32.

Recursividad - Divide y Venceras

Caso 2
Búsqueda del máximo (o del mínimo).
Este es un algoritmo en el cual se divide el vector en dos sublistas y a su vez cada una en dos sublistas mas hasta llegar al 1 solo dato (siempre iniciando por la izquierda), luego se compara con el dato de la derecha.



encontrarMaximo.cpp

Pontificia Universidad JAVERIANA Colombia

Complejidad del algortimo

Encontrar el mayor con divide y venceras vs búsqueda lineal

La complejidad del algoritmo de búsqueda del mayor está dada por la fórmula **Round(**1,4 *Ln (**N**)+1**)** , donde **n** es el número de posiciones del arreglo. Es decir en un arreglo de 16 posiciones, el número máximo de iteraciones posibles para encontrar un número es **1,4 *Ln (16)+1=5**, si el arreglo tiene 4096 posiciones, el número máximo de iteraciones será 13 porque **1,4 *Ln (4096)+1**=13

Si lo comparamos con el algoritmo de búsqueda lineal, este tiene una complejidad (**n)**, esto quiere decir que el número máximo de iteraciones esta dado por el número de posiciones del arreglo.

Es decir si el arreglo es de 16 posiciones, el número máximo de iteraciones es 16, si el arreglo es de 4096, el número máximo de iteraciones será 4096.

Recursividad

Caso 3

Palíndromo: Un palíndromo es una palabra o frase que se lee igual de izquierda a derecha o viceversa (omitiendo los espacios).

palindromo.cpp

```
#include <iostream>
#include <string>
using namespace std;
string quitarEspacios(string cadena){
  string aux="";
  for (int i=0;i<cadena.size();i++){
     if (cadena[i]!=' ‘)
        aux+=cadena[i];
     }
   return aux;
}
bool esPalindromo(string cadena,int izq, int der ){
   bool r=false;
   if ( izq<=der  )
      if  (cadena[izq]==cadena[der] )
            r=esPalindromo (cadena,izq+1,der-1);
      else
             r=false;
   else
     r=true;
   return r;
}
int main(){
   bool r;
   string cadena;
   cout <<" ingrese cadena ";
   getline (cin,cadena);
   cadena=quitarEspacios(cadena);
   r=esPalindromo (cadena,0,cadena.size()-1);
   if (r)
      cout<<"La frase es un palindromo"<<endl;
   else
     cout<<"La frase NO es palindromo"<<endl;
   return 0;
}
```

Pontificia Universidad JAVERIANA Colombia