Apuntes OIFem II Nivel 1

Recursión

La recursión...

es un proceso que se llama a así mismo. De este modo va simplificando el problema, hasta llegar a la solución. Partes principales de la recursión:

1. Caso base, este caso lo conocemos y a partir de este podemos sacar todo el resto

2. Una función/una operación para a partir del anterior/siguiente valor se pueda calcular el número buscado

Esto de momento suena muy abstracto, pero en cuanto se ven unos ejemplos es más fácil de visualizar.

Ejemplo contar orejas de conejos:

UPodemos hacer un programa que cuente las orejas de n conejos.

1.caso base n=1 orejas=2;

2.caso general n orejas=n+orejas(n-1)

Obviamente se podría calcular como 2*n, pero es una manera de visualizar lo que hace nuestra función recursiva.

```
int orejas (int n){
    if(n==1)return 2;//caso base
    else return orejas(n-1)+2;//caso general
}
```

Ejemplo con problema:

El mismo problema nos da los pasos recursivos que tenemos que dar. 1 es el caso base y lo que hacemos es recorrer la recursión y contar los pasos que hacemos en todo momento.

Link al problema

```
#include <iostream>
using namespace std;
int c(long long k){
  if(k<=1)return 1;// caso base
  if(k%2==0)return c(k/2)+1;
```

```
  else return c(3*k+1)+1;
}
int main ()
{ int n;
cin>>n;
long long k;
while(n--){
  cin>>k;
  cout<<c(k)<<"\n";
}
  return 0;
}
```

Factoriales:

Un factorial de un número n es la multiplicación de 1 por 2 por 3... por n-1 por n.

1.caso base factorial(1)=1

2.caso general factorial(n)=n*factorial(n-1)

```
int factorial (int n){
    if(n==1)return 1;//caso base
    else return factorial(n-1)*n;//caso general
}
```

Números de Fibonacci:

La secuencia de Fibonacci es la siguiente: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, etc. En términos generales:

1.caso base: f(0)=0 y f(1)=1

2.para el resto de los números f(n)=f(n-1)+f(n-2)

Con ayuda de la recursión podemos escribir un programa que calcule los números de la secuencia de Fibonacci.

Link visuaclización del código

```
int fibonacci(int n){
    if(n==0)return 0;//caso base 1
    if(n==1)return 1;//case base 2
    return fibonacci(n-1)+fibonacci(n-2);//caso general
}
```

Optimización: Memorización

Hay muchas veces que al hacer recursión llegamos al mismo número varias veces y por tanto calculamos la función varias veces y nos vale con una. Guardándonos el resultado podemos ahorrarnos mucho tiempo.

Por ejemplo para calcular Fibonacci(4) haríamos lo siguiente

1. Fibonacci(4)=Fibonacci(3)+Fibonacci(2) // aquí abrimos para calcular 3 y 2

2. Calculando Fibonacci 3 tendríamos que volver a calcular Fibonacci 2, que nos lo habríamos ahorrado si lo hubiéramos calculado antes

3. Si seguimos haciendo esto veremos que este caso se repite muy a menudo. Si quieres puedes mirar la visualización en el link de arriba, que cuenta el número de repeticiones.

¿Cómo guardamos esta información?

Creamos un vector¡int¿memo(n+1,-1) Cuando entramos en la función preguntamos si memo[i]!=-1 y por tanto ya esta calculado, si es el caso entonces devolvernos ese número. En caso contrario calculamos el número y lo guardamos.

```
vector<int>memo;
int fibonacci(int n){
    if(n==0)return 0;
    if(n==1)return 1;
    if(memo[n]!=-1)return memo[n];
    return memo[n]=fibonacci(n-1)+fibonacci(n-2);
}
```

Recursión múltiple:

Podemos tener varias funciones, que se llamen entre ellas de manera recursiva. El funcionamiento es el mismo que el de la recursión simple, pero ahora tenemos varias funciones. La función a llama a la función b y la función b llama a la función a y así sucesivamente hasta llegar al caso base con el que se acaba la recursión. (Pueden ser más de dos funciones que se llamen entre si, pero de momento veremos ejemplos con máximo dos.)

Problema práctico: Poniendo la mesa

Los invitados están al llegar y la mesa sigue sin poner. En la mesa de la cocina, tus padres han preparado ya todo lo que hay que trasladar: pilas de platos, cubiertos y copas. Por delante, un montón de paseos de la cocina al salón y muy poco tiempo.

Como las prisas en este caso no son buenas (que haya que pararlo todo para recoger del pasillo los pedazos de una copa rota puede ser desastroso) la única solución es paralelizar el trabajo. Tu hermano pequeño es el candidato perfecto para ayudarte. Empezaréis por llevar todas las copas. Como son delicadas, tu hermano las llevará de una en una. Y para que se sienta "mayor" le has dicho que tú harás lo mismo: las llevarás también de una en una a no ser que el número de copas que queden en la cocina sea par. En ese caso en lugar de una, llevarás dos. Si el primer paseo lo da tu hermano y os vais alternando los viajes, ¿cuántos necesitaréis para llevar todas las copas?

Solución: Montaremos una recursión múltiple:

-int turnohermano(int n)

-int miturno(int n)

Siempre empieza el hermano y os vais alternado hasta que no queden vasos, es decir llegar al caso base n=0. **Link al problema**

```cpp
#include <iostream>
#include <vector>
using namespace std;
int miturno(int n);//necesitamos incluirlo aquí para poder usar la recursión multiple

int turnohermano(int n){
  if(n<=0)return 0;//caso base
  return 1+ miturno(n-1);//da el paseo y el siguiente te toca a ti
}

int miturno(int n){
  if(n<=0)return 0;//caso base
  if(n%2==0)return 1+turnohermano(n-2);//si son pares llevas dos y le toca a tu hermano
  return 1+turnohermano(n-1);//si son inpares llevas una y le toca a tu hermano
}

int main(){
  int copas;
  while(cin>>copas){
    if(copas==0)break;
    cout<<turnohermano(copas)<<"\n";
  }
```

```
23      return 0;
24  }
```