UCAB UNIVERSIDAD CATÓLICA ANDRÉS BELLO

Algoritmos y Estructuras de Datos

Oscar Meza

omezahou@ucab.edu.ve

Sobre el proyecto 1

PROYECTO 1: Asignación de horarios

El objetivo de este proyecto es la asignación de horarios a diferentes secciones de clases conformadas, cada una, por un profesor y varios estudiantes.

Descripción general:

La idea del proyecto es realizar **un programa en C++ para asignar las horas de clase en un día específico (por ejemplo, lunes), a varias secciones de clase**. Una sección de clase se corresponde con una sección de clase de la UCAB, es decir, una sección está conformada por un profesor y un conjunto de estudiantes. La sección corresponde a una materia, por ejemplo, “Algoritmos y Estructuras de Datos INFO-02002”.

Cada sección tiene un conjunto de horas de dictado de clases por semana, que se distribuyen en a lo sumo tres bloques de horas; por ejemplo, “Algoritmos y Estructuras de Datos INFO-02002” tiene tres bloques de horas, a saber, 2 horas de teoría, 2 horas de práctica y 2 horas de laboratorio. Otra sección de otro curso, por ejemplo “Matemáticas Discretas INFO-02000” tiene 2 horas de teoría, otras 2 horas de teoría y 1 hora de práctica.

Una sección de clase cualquiera, tendrá a lo sumo tres bloques de horas. Por ejemplo, “Algoritmos y Estructuras de Datos INFO-02002” tiene los bloques 2, 2, 2; mientras que “Matemáticas Discretas INFO-02000” tiene los bloques 2, 2, 1. Otra sección de algún otro curso tendrá los bloques 2,2. Lo importante es que supondremos que una sección tendrá 1, 2 ó 3 bloques de horas. Las combinaciones de bloques permitidas son 2,2,2 ó 2,2,1 ó 1,1 ó 3 ó 2,1 ó 1,3 ó 2.

El objetivo final del proyecto es asignar un horario a los bloques de cada sección de clase en un día determinado.

RECORDEMOS el uso de archivos header .h para reforzar el ocultamiento de datos

Hagamos la implementación del tipo conjunto de enteros **con header files.**

Utilice un vector para almacenar un conjunto de enteros y defina su tipo como:

typedef std::vector<int> Conjunto_int;

ó

using Conjunto_int = std::vector<int> ;

Las operaciones que implementaremos:

- **Insertar elemento**
- **Verificar si un elemento pertenece al conjunto.**
- **Imprimir los elementos de un conjunto**
- **Unión de dos conjuntos**

Archivo Conjunto.h:

```
#include <vector>
using std::vector;
typedef vector<int>     Conjunto_int;
Conjunto_int     insertar(Conjunto_int v, int i);
bool             pertenece(Conjunto_int v, int i);
Void             imprimir(Conjunto_int);
Conjunto_int     unir(Conjunto_int,Co njunto_int);
bool             es_vacio(Conjunto_int);
```

Archivo Conjunto.cpp: (note que no es necesario incluir #include <vector> pues está en el .h)

```
#include "Conjunto.h"

Conjunto_int insertar(Conjunto_int v, int i){
   // Verificar que no esté para poderlo insertar (definición de conjunto)
   for (int j=0; j<v.size(); j++){
      if (i==v[j]) return v;
   }
   v.push_back(i);
   return v;
}
```

Continúa...

```
bool pertenece(Conjunto_int v, int i){
    for (int j=0; j<v.size(); j++){
        if (i==v[j]) return true;
    }
    return false;
}

void imprimir(Conjunto_int conj){
    for (int j=0; j<conj.size(); j++)
        std::cout << conj[j] << " " ;
}
```

```
bool es_vacio(Conjunto_int conj){
   if (conj.empty()) return true;
   else return false;
}

Conjunto_int unir(Conjunto_int A, Conjunto_int B){
   Conjunto_int   result;
   for (auto i : A)
     result.push_back(i);
   for (auto i : B)
     if (! pertenece(A,i))
       result.push_back(i);
   return result;
}
```

Archivo prueba_conjunto.cpp (note que por encapsulamiento de datos no tenemos que conocer como se implementa Conjunto_int, solo utilizar sus operaciones):

```
#include <iostream>
#include "Conjunto.h"

int main(){
    Conjunto_int conj;
    conj = insertar(conj,0);
    conj = insertar(conj,1);
    conj = insertar(conj,2);
    imprimir(conj);
    int i=3;
    std::cout << "¿Está " << i  << " en el conjunto?: "
      <<  std::boolalpha << pertenece(conj,i)
      << "   Es vacio?: " << es_vacio(conj);

    Conjunto_int conj1;
    conj1 = insertar(conj1,0);
    conj1 = insertar(conj1,4);
    conj1 = insertar(conj1,5);
    Conjunto_int c = unir(conj,conj1);
    imprimir(c);
}
```

Archivos de texto de lectura y escritura

Como crear un Archivo de texto de salida (crear un archivo de texto):

```
#include <iostream>
#include <fstream>
#include <string>
int main() {
    int entero;
    std::string filename;
    std::cout
      << " Nombre del archivo: ";
    std::cin >> filename;
    // crear archivo
    std::ofstream out(filename); //crea out y lo
                    // asoci al al archivo (lo abre)
    if (out.is_open())  {     // si se abrio bien el archivo
        // colocar varios enteros y terminar con un caracter
        while (std::cin>>entero){
         // termina si coloco un carácter,
         // lo hace por un error  al leer un caracter.
         //Habría que  "resetear "cin" para utilizarlo después
              out << entero << " ";
          }
    }   else    std::cout   << "No se pudo escribir\n";
    out.close();
    return 0;
}
```

Como acceder a un Archivo de texto de entrada (leemos un archivo):

```
std::cout << " Coloque el nombre del archivo: ";
std::cin >> filename;
std::ifstream in(filename);
int value;
if (in.is_open()) {  // verificar archivo abierto
   while (in >> value)  // leer hasta fin de archivo
        std::cout << value << "\n";
}
else   std::cout << "No se puede abrir el archivo\n";
in.close();
```

Como acceder a un Archivo de texto de entrada (leemos un archivo):

Se puede utilizar:

string sa;

```
getline(in, sa)
```

Y obtener una línea completa del archivo

Recursividad

Funciones recursivas:

- **¿Cómo hacer una función que calcule recursivamente n! en C++?**

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot (n-1)! & \text{otherwise.} \end{cases}$$

```
#include <iostream>
/*
 *  factorial(n)
 *       Calcula n!
 *       Devuelve el factorial de n. (n debe ser >=0)
 */
int factorial(int n) {
   if (n == 0)        return 1;   //caso base
   else        return n * factorial(n - 1);
}
int main() {
   // Probemos la función factorial recursiva
   std::cout << " 6! = " << factorial(6) << '\n';
}
```

HABLEMOS DE RECURSIÓN

Una definición recursiva de un objeto en matemáticas y en computación, **consiste en definir el objeto en términos de él mismo. Por ejemplo una expresión aritmética** usando los operadores binarios +, - y el unario -.

- **Una expresión aritmética se define recursivamente como:**
  - una cadena de caracteres (letras y números) comenzando con letra es una expresión aritmética. (base de la definición recursiva, una variable)
  - si e1 y e2 son expresiones aritméticas entonces (e1+e2), (e1-e2) y (– e1) son expresiones aritméticas
  - cualquier expresión aritmética se obtiene aplicando las reglas anteriores un número finito de veces
- **Podemos formar expresiones tan grandes como queramos veamos….**

- Por ejemplo:

  - **x** e **y** son expresiones aritméticas
  - **(x+y)** es una expresión aritmética obtenida de las dos anteriores, aplicando el operador +
  - **((x+y)-y)** es una expresión aritmética obtenida de las anteriores y aplicando el operador -, y así sucesivamente…
  - Aplicando las reglas podemos conseguir infinitas operaciones aritméticas

- La **solución** de un problema es **recursiva** si como parte de la solución del problema está la resolución de varios **sub-problemas "más pequeños" del mismo tipo que el problema original.**
- Es crucial que al aplicar la solución recursiva, **ésta culmine en algún momento**, por eso hablamos de sub-problemas más pequeños.
- Por ejemplo:

  El factorial de un número entero no negativo N se define como:

  $$N! = N*(N-1)*\ldots*2*1 \quad \text{con} \quad 0!=1$$

  Podemos expresar N! de manera recursiva:

  $$N! = N* (N-1)! \quad \text{con } 0! = 1 \text{ (base de la recursión)}$$

- N! = N*(N-1)!, para calcular N! aplicamos la solución recursiva. Aplicamos la misma definición recursiva para calcular (N-1)!.
- Así (N-1)! = (N-1)*(N-2)!, y (N-2)!= (N-2)*(N-3)! Hasta llegar a la base 1! = 1* 0!
- Y como el caso base 0! sabemos que es igual a 1. Tendremos 1! = 1*1
- Y luego, para hallar el resultado, vamos en sentido contrario ("subiendo" en la recursión) para calcular 2! = 2*1! = 2*1 = 1, luego 3!, y así sucesivamente hasta llegar a calcular N!.

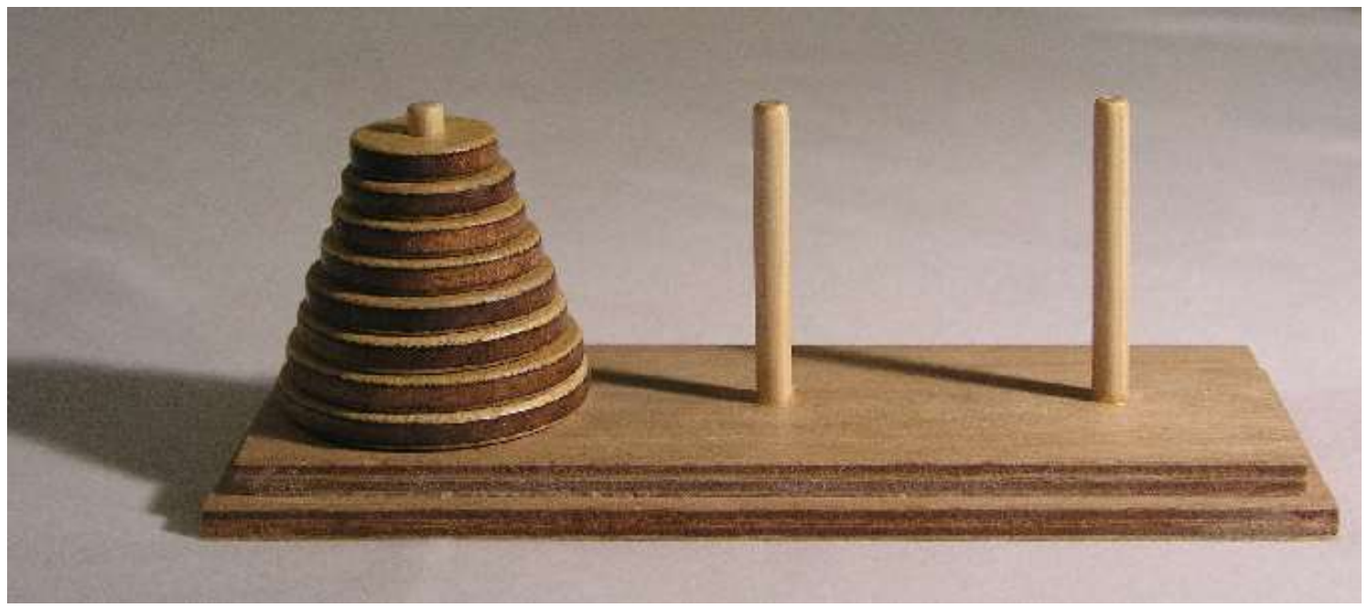
Una solución recursiva se sustenta en dos hechos fundamentales:

1. Existe un conjunto de **casos bases (problemas base del mismo tipo)** que no se resuelven de manera recursiva sino directa.

2. **Los sub-problemas "más pequeños"** tienen menos datos. Lo importante es que **los sub-problemas van convergiendo en forma estricta a algunos de los casos base**. Cuando decimos en forma estricta es para evitar que la solución del problema quede expresada en función de la solución del mismo problema con los mismos datos y se crea así una **solución cíclica**, que se devuelve al problema original y no lo resuelve.

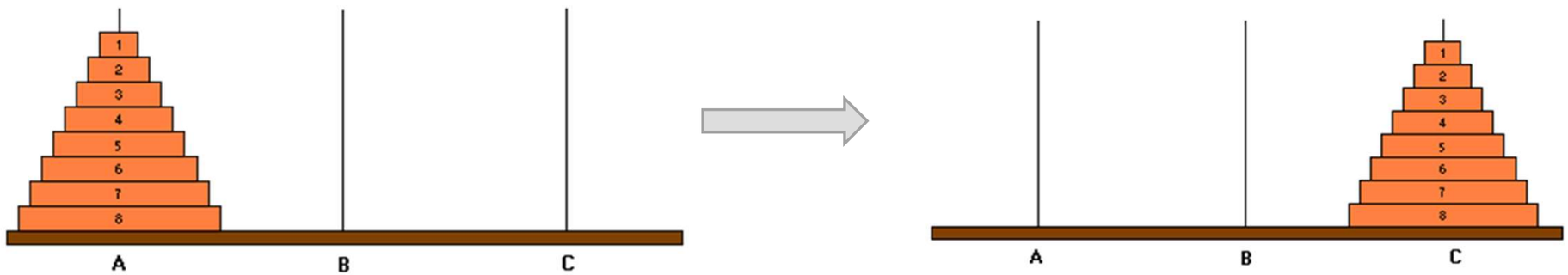
En el caso del factorial vemos que el subproblema (N-1)! Es más pequeño pues converge al caso base en forma estricta, en este caso 0!=1

Veamos un problema clásico que tiene una **solución recursiva** que surge **de manera natural** y sin embargo una **solución iterativa** no es fácil de conseguir: **Las Torres de Hanoi**



https://upload.wikimedia.org/wikipedia/commons/0/07/Tower_of_Hanoi.jpeg

El juego consiste en pasar todos los platos desde el poste origen (es decir, el que posee la torre con los platos) al poste destino y que queden en el mismo orden. Y utilizando el otro poste como auxiliar.



- Se debe seguir las reglas:
  - Sólo se puede mover un disco a la vez de un poste a otro
  - Un disco no puede estar sobre uno de menor diámetro
  - Sólo se puede mover el disco que se encuentre arriba en cada poste.

Suponga que se quiere mover **la torre de N platos de A a C y B como auxiliar**.

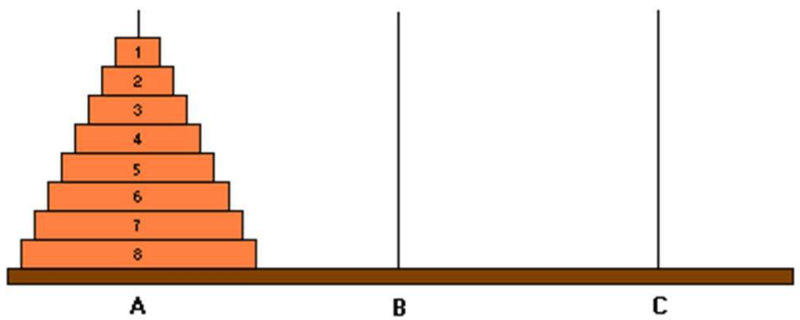
Solución recursiva:

Mover resursivamente la torre de N platos de A a C y B como auxiliar:

1) Mover recursivamente N-1 platos de A a B y C como auxiliar

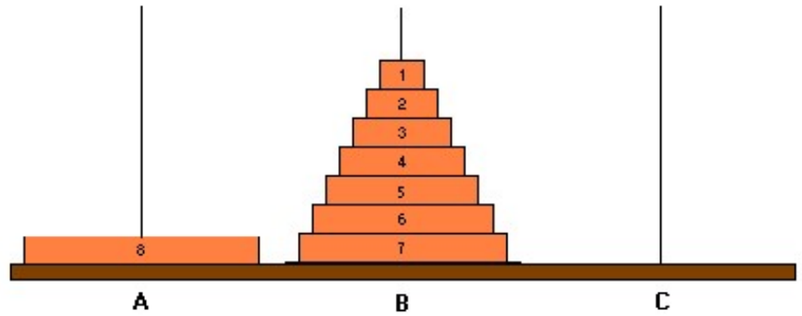
2) Mover el plato N (el más grande) de A a C

3) Mover recursivamente la torre de N-1 platos de B a C y A como auxiliar

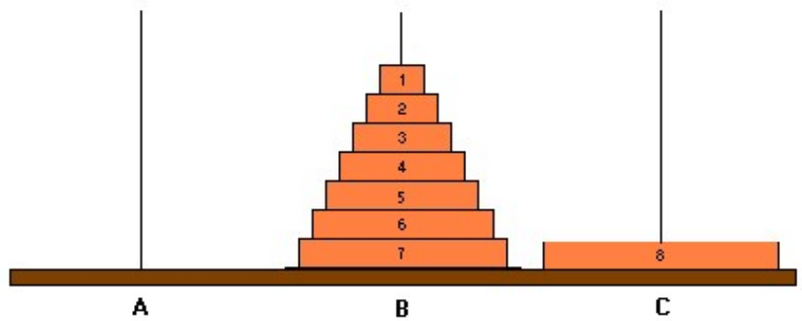


https://carlosdelgado90.wordpress.com/torres-de-hanoi/

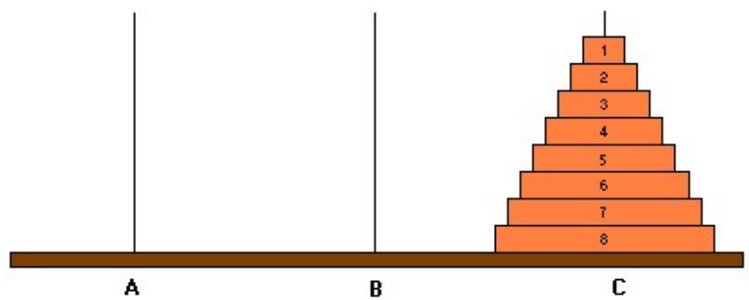
- **Primer paso recursivo: mover N-1 platos de A a B**



- **Segundo paso:**



- **Tercer paso recursivo: mover N-1 platos de B a C**



¿Cuál sería el caso base?

Método en C++ que implementa la solución recursiva, N es el número de platos:

```
void tHanoi (int N, string origen , string destino, string auxiliar )
{       if ( N == 1)
                std::cout << "mover desde “<<origen<<" a “<<destino<<‘\n’;
        else  {  tHanoi ( N - 1 , origen, auxiliar, destino);
                 std::cout << "mover desde “<<origen<<" a “<<destino<<‘\n’;
                tHanoi ( N - 1 , auxiliar, destino, origen);
        }
}
```

Llamada: tHanoi(3, ”origen” , ”destino” , ”auxiliar”);

El programa anterior imprime para N=3:

mover desde origen a destino

mover desde origen a auxiliar

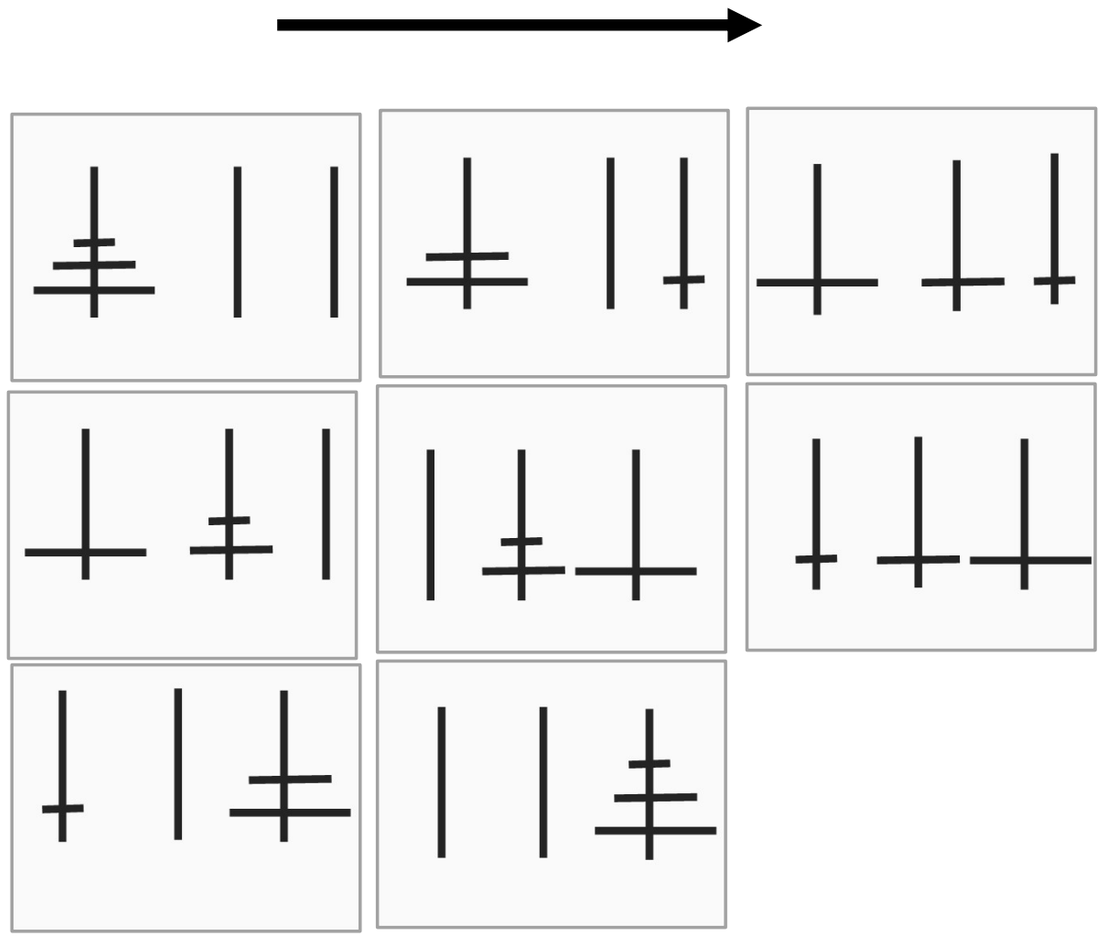
mover desde destino a auxiliar

mover desde origen a destino

mover desde auxiliar a origen

mover desde auxiliar a destino

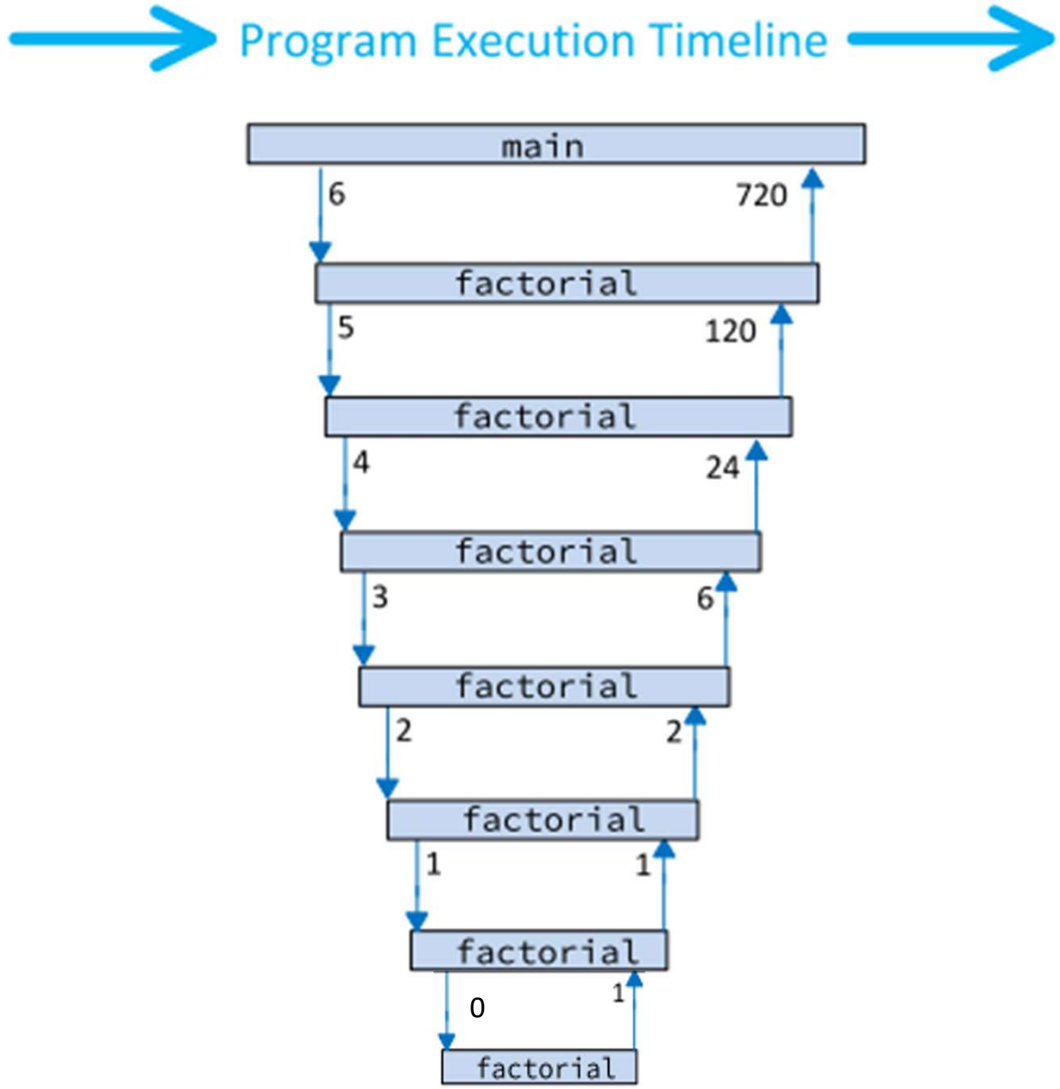
mover desde origen a destino



¿Cómo se ejecuta un programa recursivo en el computador?:

En llamadas **sucesivas** a varias funciones, el sistema va **guardando apilados (en una pila ó stack) los estados (activation record) de las sucesivas llamadas** y luego al terminar la ejecución de una llamada a una función, se borra su estado (desaparecen las variables locales de ese estado) y se regresa al estado de la función que lo llamó.

Por ejemplo: calcular **6!**



Mientras se pueda evitar la recursión mejor es, pues crea mucho **overhead** (tiempo indirecto de cómputo con la generación del stack de estados y uso adicional de memoria).

Sólo utilizarla en los casos que lo ameriten como veremos a lo largo del semestre.

Cuando un algoritmo recursivo hace sólo una llamada recursiva decimos que es un algoritmo con **recursión de cola** y es fácil de convertirlo en un programa iterativo equivalente.

```
long factorial( int n )
{   // n debe ser mayor o igual que cero

   if ( n <= 0 )
      return 1;
   else
      return n*factorial(n-1);

}
```

Cuando un algoritmo recursivo hace sólo una llamada recursiva decimos que es un algoritmo con **recursión de cola** y es fácil de convertirlo en un programa iterativo equivalente.

```
long factorial( int n )
{   // n debe ser mayor o igual que cero

   if ( n <= 0 )
      return 1;
   else
      return n*factorial(n-1);

}
```

```
long factorial( int n )
{   // n debe ser mayor o igual que cero

   int fact = 1;
   if ( n <= 0 )    ; // no haga nada
   else    {    int i = 1;
               while (i <= n)
                              fact = fact * i++ ;
            }
   return fact;
}
```

Hagamos una función recursiva que imprima un vector de enteros. Se pasa el vector pero luego se necesitará una función auxiliar

```
#include <iostream>
#include <vector>
using namespace std;
void imprimir_rec (int i, vector<int> v){
        if (i > (v.size()-1)) return;
        std::cout << v[i] << "  "  ;
        imprimir_rec (i+1, v) ;
}
```

```
void imprimir (vector<int> v){
        int i = 0;
        imprimir_rec(i, v);
       std::cout << '\n';
}
int main( ){
    vector<int> v {1,2,3,4};
    imprimir(v);
}
```

¿ Lo podemos hacer sin la función auxiliar?.

Si. Imprimiendo el primer elemento y copiando el resto en un vector de tamaño uno menos

```
void imprimir1 (vector<int> v){
        if (v.empty()) return;
        std::cout << v[0] << "  "  ;
        vector<int> v1;
        for (int i=1; i< v.size(); i++)
           v1.push_back(v[i]) ;
        imprimir1 (v1);
}
```

Son malísimos estos algoritmos porque hacen sucesivas copias del vector (lo solucionaremos cuando veamos apuntadores….)

Hagamos juntos una función recursiva que imprima el vector en orden inverso.

Puede usar una función recursiva auxiliar.

```
#include <iostream>
#include <vector>
using namespace std;
void imprimir_rec_inverso (int i,
          const vector<int>& v){
    if (i > (v.size()-1)) return;
    imprimir_rec_inverso (i+1, v) ;
    std::cout << v[i] << " " ;
}
```

```
void imprimir_inverso
                (const vector<int> &v){
    imprimir_rec_inverso(0, v);
    std::cout << '\n';
}
int main( ){
    vector<int> v {1,2,3,4};
    imprimir_inverso(v);
}
```

Otra solución (que supone que tenemos acceso al último elemento del vector, que en este caso en efecto es así)

```
#include <iostream>
#include <vector>
using namespace std;
void imprimir_rec_inverso (int i,
          const vector<int> v){
    if (i < 0) return;
    std::cout << v[i] << " " ;
    imprimir_rec_inverso (i-1, v);
}
```

```
void imprimir_inverso (const vector<int> v){
    imprimir_rec_inverso(v.size()-1, v);
    std::cout << '\n';
}
int main( ){
    vector<int> v {1,2,3,4};
    imprimir_inverso(v);
}
```

Hagamos una función recursiva que imprima todas las permutaciones de los elementos de un vector de enteros de tamaño N.

El vector contiene: 1, 2, 3, 4, ... , N

```
void permutaciones(vector<int> A){
    if (A.empty()) return;
    permutaciones(0,A);
}
```

se llama en main() :

```
 permutaciones(Vector)
```

```
void permutaciones(int n, vector<int> A){
    int temporal;
    if (n==A.size()) print(A);
    else
     for (int i = n ; i < A.size(); i++){
        temporal=A[n];
        A[n]=A[i];
        A[i]=temporal;
        permutaciones(n+1,A);
        temporal=A[n];
        A[n]=A[i];
        A[i]=temporal;
     }
}
```

¿Cómo sería un programa recursivo en C++ para calcular el N-ésimo número de Fibonacci?

Donde $F_0 = 0$, $F_1 = 1$, $F_n = F_{n-1} + F_{n-2}$ *para* $n>=2$

Hagámoslo juntos…….

Donde $F_0 = 0$, $F_1 = 1$, $F_n = F_{n-1} + F_{n-2}$ *para* $n>=2$

Un algoritmo recursivo sería:

```
/* *
*     fib(n):
*         n debe ser un entero no negativo
*         Devuelve el número de Fibonacci número n
**/
long    fib( int n )
{   if( n <= 1 )
            return n;
    else
            return fib( n - 1 ) + fib( n - 2 );
}
```

este algoritmo es exponencial función de n

Un algoritmo iterativo sería:

```
long fibonacci_itr( int n )
 {  // n debe ser >= 0
    int fn_1 = 1, fn_2 = 0, fib = 0 ;
    if (n==0 || n==1) return n;
    for (int i = 2; i<=n; i++) {
     fib  =  fn_1  +  fn_2;    fn_2 = fn_1;     fn_1 = fib;
    }
    return fib;
 }
```

Mejor en tiempo (n-1 sumas) y espacio que el recursivo....

PREGUNTAS???