Unidad 1

Fundamentos de Programación Competitiva

Logro de sesión

Al finalizar la sesión, el estudiante comprenderá Análisis de Complejidad Algorítmica

Semana 2

Análisis de Complejidad Algorítmica

Contenido:

- Análisis de Complejidad Algorítmica
- Notaciones Asintóticas
- Algoritmos recursivos

Algoritmo

- Un algoritmo es un método para resolver un problema, López et al. (2009) definen algoritmo como “un conjunto de pasos que, ejecutados de la manera correcta, permiten obtener un resultado (en un tiempo acotado)”.

- Pueden existir varios algoritmos para resolver un mismo problema. Cuando se estudian los algoritmos es importante analizar tanto su diseño como su eficiencia.

Algoritmo

Se buscan:

Métodos y/o Procedimientos

Secuencia finito de instrucciones

resuelven

Problema

Algoritmo

Propiedades de Algoritmos:

1. Entradas y salidas definidas.
2. Es preciso, evitar ambiguedades sobre el orden exacto de los pasos a seguir.
3. Es determinista, producir el mismo resultado con diferentes datos de entrada.
4. Es finito, contiene un número finito de pasos.
5. Es efectivo, cada paso se debe llevar en un tiempo finito.

Análisis de Algoritmos

"El análisis de algoritmos pretende descubrir si estos son o no eficaces. Establece además una comparación entre los mismos con el fin de saber cuál es el más eficiente, aunque cada uno de los algoritmos de estudio sirva para resolver el mismo problema".
[Villalpando, 2003]

Eficiencia de Algoritmos

Por lo general los aspectos a tomar en cuenta para estudiar la eficiencia de un algoritmo son el **TIEMPO** que se emplea en resolver el problema y la **CANTIDAD DE RECURSOS DE MEMORIA** que ocupa. Para saber qué tan eficiente es un algoritmo hacemos las preguntas:

- ¿Cuánto tiempo ocupa?
- ¿Cuánta memoria ocupa?

Análisis de Algoritmos

- El tiempo de ejecución de un algoritmo depende de los datos de entrada, de la implementación del programa, del procesador y finalmente de la complejidad del algoritmo.

- El tiempo que requiere un algoritmo para resolver un problema está en función del tamaño n del conjunto de datos para procesar: T(n).
  [López et al., 2009].

Análisis Complejidad Algorítmica

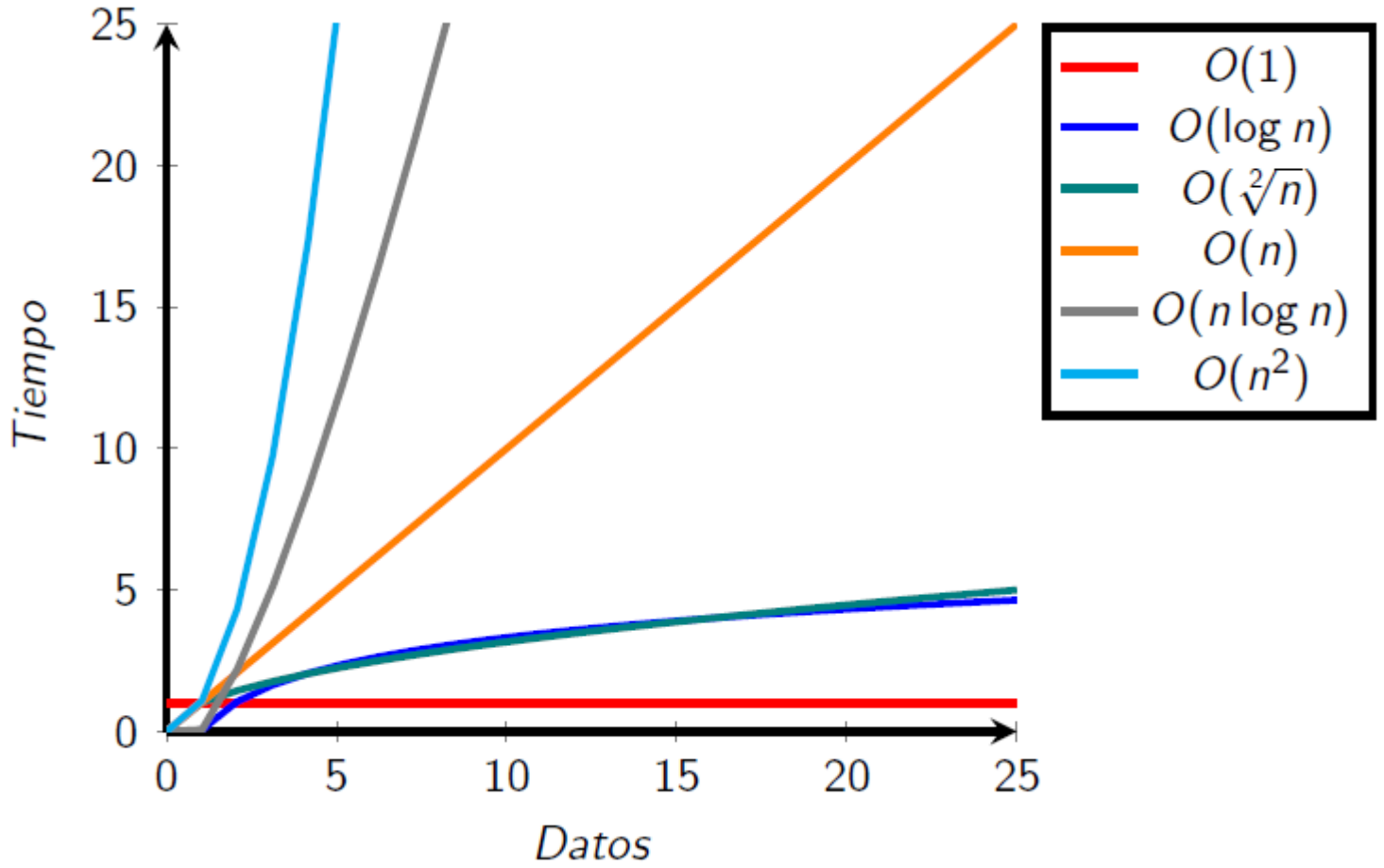
Medidas asintóticas:

Permiten analizar que tan rápido crece el tiempo de ejecución de un algoritmo cuando crece el tamaño de los datos de entrada

Notación Big O

- Big O representa el peor de los casos.
- Nos permite evaluar los términos de upper bounds, es decir proporcional a lo máximo que podrá demorar un algoritmo.

Análisis Complejidad Algorítmica



Análisis Complejidad Algorítmica

Notación Big O – Significado

- O(1): **Constante**, significa que da exactamente igual que el algoritmo tenga uno o un millón de instrucciones: insertar o recuperar un elemento siempre tarda más o menos lo mismo.

- O(n): **Lineal,** significa que el tiempo necesario para ejecutar la función es función directa y lineal del número de elementos que le pasemos.

Análisis Complejidad Algorítmica

Notación Big O – Significado

- O(log n): **Logarítmica**, por ejemplo búsqueda binaria.
- O(n log n): por ejemplo algoritmo de búsqueda Quicksort.
- O (n2): **Cuadrática**, casos poco optimizados

Análisis Complejidad Algorítmica

Tiempo de ejecución de un algoritmo

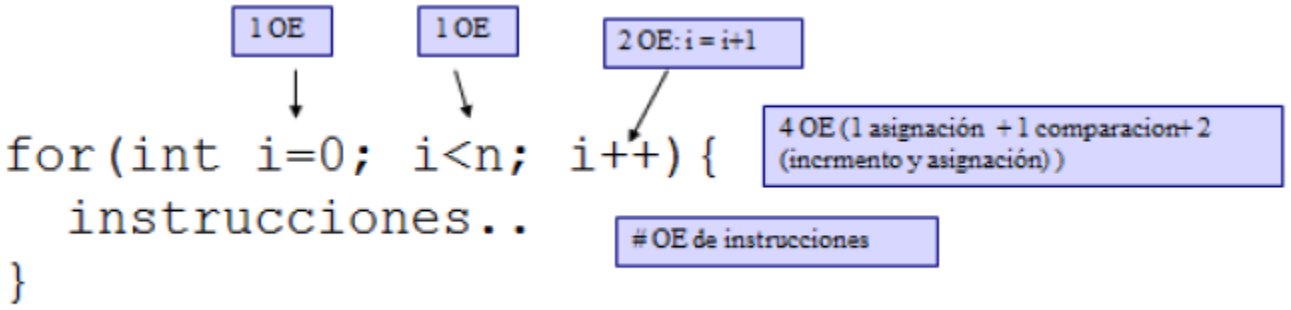
Que podemos contar?

- ❑ Operación aritmética
- ❑ Asignación a una variable
- ❑ Llamada a una función
- ❑ Retorno de una función
- ❑ Comparaciones lógicas (con salto)
- ❑ *Acceso a una estructura (arreglo, matriz, listas…)*

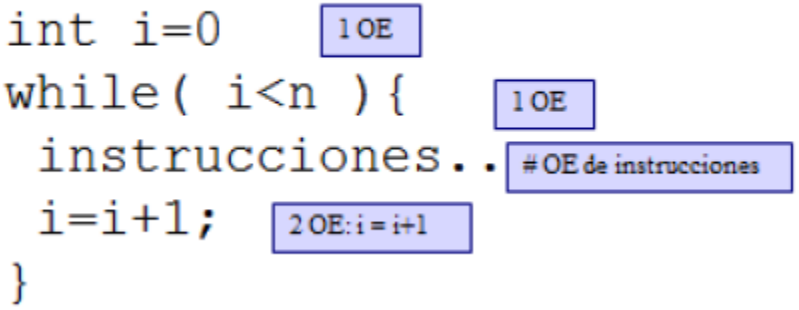
Se le llama tiempo de ejecución, no al tiempo físico, sino al número de operaciones que se llevan acabo en un algoritmo.

Análisis Complejidad Algorítmica

Donde: “OE” significa operaciones elementales



Forma equivalente del for:



Análisis Complejidad Algorítmica

¿Cuántas operaciones elementales se realizan en este algoritmo?

```
#include <cstdlib>
#include <iostream>

using namespace std;

int main(int argc, char *argv[])
{
    int ArregloOrdenado[9]={1,3,7,15,19,24,31,38,40};
    int buscado, j=0;

    cout<<"¿Que numero entero quieres buscar en el arreglo?";
    cin>>buscado;

    while(ArregloOrdenado[j] < buscado && j<9 )
      j = j+1;

    if ( ArregloOrdenado[j] == buscado)
      cout<<"tu entero si esta en el arreglo";

    else
      cout<<" lastima! no esta";
    system("PAUSE");

    return EXIT_SUCCESS;
}
```

1 OE (asignación)

1 OE (asignación)

1 OE (salida)

1 OE (entrada)

4 OE (1 acceso + 2 comparaciones+ 1 AND)

2 OE (incremento + asignación)

2 OE (acceso + comparación)

1 OE (salida)

1 OE (salida)

1 OE (pausa)

1 OE (regreso)

Algoritmos de Recursividad

Algoritmos de Recursividad

- La recursividad es una técnica de programación en la que un módulo hace una llamada a sí mismo con el fin de resolver el problema. La llamada a sí mismo se conoce como llamada recursiva.

- Un algoritmo se dice recursivo si calcula instancias de un problema en función de otras instancias del mismo problema hasta llegar a un **caso base**, que suele ser una instancia pequeña del problema, cuya respuesta generalmente está dada en el algoritmo y no es necesario calcularla.

Algoritmos de Recursividad

1. Calcular el factorial de un número

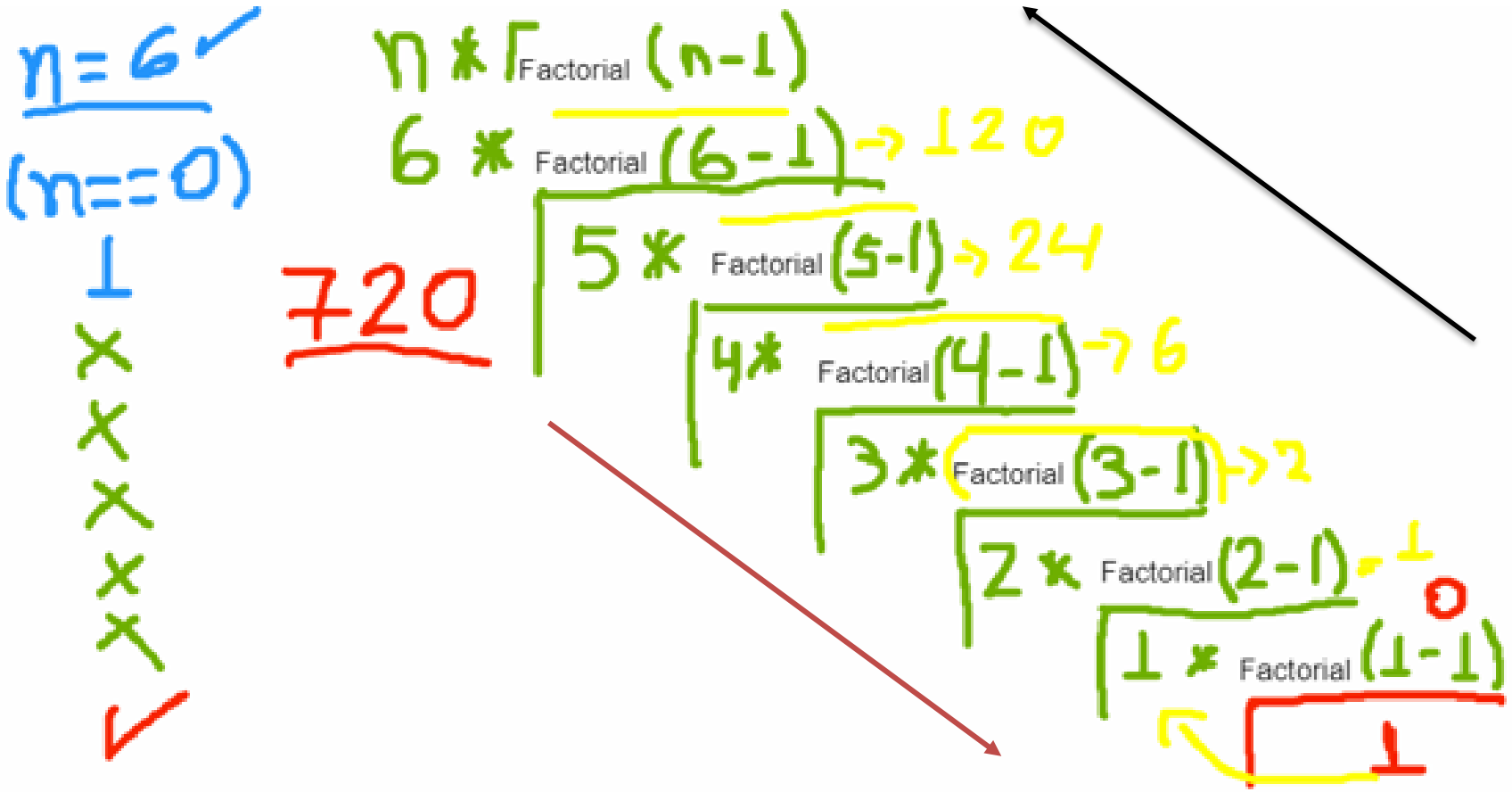
```
#include<iostream>
#include<conio.h>
using namespace std;

int factorial(int n)
{
     if (n == 0)
         return 1;
     else
        return n * factorial(n - 1);
}
```

```
int main()
{
      int n;
      int resultado;
      cout << "\nIngrese un
número: ";
      cin >> n;
      resultado =
factorial(n);
      cout << "\nEl factorial
es: " << resultado;
      getch();
      return 0;
}
```

Algoritmos de Recursividad

1. Calcular el factorial de un número – Tabulación



Algoritmos de Recursividad

2. Invertir un número

```
#include"pch.h"
#include<iostream>
#include<cstdlib>
using namespace std;

void invertir(int nro) {
    cout << nro % 10;
    if (nro > 10) invertir(nro / 10);

}
int main(void) {
    system("color 0a");
    int nro;
    cout << "\n\t\t[      RECURSIVIDAD
]\n";
 cout << "\t\t--------------------
----\n\n";
    cout   <<   "   EJERCICIO   4:
Invertir un numero " << endl <<
endl;
    do {
        cout << " INGRESE NUMERO:
";
        cin >> nro;
        if   (nro   <   0)   cout   <<
"\nINGRESE UN NUMERO ENTERO Y
POSITIVO... \n";
    } while (nro < 0);
    cout << "\n NUMERO:" << nro;
    cout << "\nINVERTIDO:";
    invertir(nro);
    cout << endl << endl;
    return 0;
}
```

Muchas gracias!!!