CLUB DE PROGRAMACIÓN COMPETITIVA UMSA

# INF-143
# TALLER DE PROGRAMACIÓN

FEBRERO 2022

# Presentación

Univ. Grover Osvaldo Rodriguez Apaza

**Carrera de Informática**
**Facultad de Ciencias Puras y Naturales**

18 de Febrero de 2022

# Contenido

1 Funciones
- ¿Qué es una Función?
- Varias Funciones
- Ejemplo

2 Recursividad
- ¿Qué es Recursividad?
- Ejemplo 1
- Ejemplo 2
- Ejemplo 3

3 Exponenciación Rápida
- Calcular potencias
- Calcular potencias 2
- Complejidad

4 Exponenciación Matricial
- Matrix
- Matrix 2
- Complejidad

# 1

# Funciones

# ¿Qué es una Función?

Una Función es un segmento de código fuera de la función principal que cumple una tarea específica, además puede o no retornar (un tipo de dato, estructura de datos).

**Ejemplo:**

Se desea contruir una función para sumar dos números enteros.

```
int suma_dos_numeros(int a, int b){
    int suma = a + b;
    return suma;
}
```

**Complejidad:** $O(1)$

# Varias Funciones

Así como en la función principal se puede llamar a otras funciones, de igual manera se puede llamar en la misma función a otras funciones

**Ejemplo 1:**

Hallar el valor de $u$ en la siguiente expresion, siendo $x$ la entrada al programa.

$$u = x \cdot f(x!)$$

Donde:

$$f(x) = 4x + 5$$

# Ejemplo

```cpp
#include <bits/stdc++.h>
using namespace std;
int f(int x){
    return 4 * x + 5;
}
int fact(int x){ // calcula x!
    int ans = 1;
    for(int i = 1; i <= x; i++)
        ans *= i;
    return ans;
}
int sol(int x){ // calcula el valor de u
    int u = x * f(fact(x));
}

int main(){
    cout << sol(2) << '\n';
    return 0;
}
```

**Complejidad:** $O(n)$

# 2

# Recursividad

# ¿Qué es Recursividad?

Una función se dice que es recursiva **si y solo si**, se vuelve a llamar a si misma en la misma función.
Una función recursiva tiene dos partes:

- Caso base
- Caso recursivo

# Ejemplo 1

## Imprimir Números

**Descripción**

Dado un número $n$ por teclado, mostrar los primeros $n$ números en orden descendente.

**Entrada**

La entrada consiste de un número entero $n$ $(1 \leq n \leq 100)$.

**Salida**

La salida consiste de una línea con los primeros $n$ números en orden descendente.

| Ejemplo Entrada | Ejemplo Salida |
|---|---|
| 5 | 5 4 3 2 1 |

# Solución al ejemplo 1

## Algoritmo 1 (Iterativo)

```cpp
#include <bits/stdc++.h>
using namespace std;

int main(){

    int n = 5;
    for(int i = n; i >= 1; i--)
        cout << i << ' ';
    return 0;
}
```

**Complejidad:** $O(n)$

# Análisis de la solución al ejemplo 1

**Caso Base**

El último número a mostrar es el $1$, entonces cuando $n = 0$, debería terminar la recursión

**Caso Recursivo**

Como hay que mostrar de manera decreciente, para pasar del estado $n$ al siguiente estado, es $n - 1$, ya que si $n = 5$, entonces $n - 1 = 4$.

Expresado de otra manera:

$$sol(n) = \begin{cases} return & \text{si } n = 0 \text{ caso base} \\ print(n), sol(n-1) & \text{caso recursivo} \end{cases}$$

# Solución al ejemplo 1

## Algoritmo 1.1 (Recursivo)

```cpp
#include <bits/stdc++.h>
using namespace std;

void sol(int n){
    if(n == 0){
        return;
    }
    cout << n << ' ';
    sol(n - 1);
}

int main(){

    int n = 5;
    sol(n);
    return 0;
}
```

**Complejidad:** $O(n)$

# Ejemplo 2

## Factorial

**Descripción**

Dado un número $n$ calcular el factorial de $n$

$n! = 1 \cdot 2 \cdot 3 \cdot 4 \ldots n$

**Entrada**

La entrada consiste de un número entero $n$ $(1 \leq n \leq 10)$.

**Salida**

La salida consiste de una línea, el factorial de $n$

| Ejemplo Entrada | Ejemplo Salida |
|---|---|
| 3 | 6 |

# Análisis de la solución al ejemplo 2

**Caso Base**

Matematicamente el factorial de $0$ es $1$, entonces:

- cuando $n = 0$, return 1

**Caso Recursivo**

el factorial de un numero esta definido matemáticamente como $n! = (n-1)! \cdot n$

Expresado de otra manera:

$$fact(n) = \begin{cases} 1 & \text{si } n = 0 \text{ caso base} \\ fact(n-1) \cdot n & \text{caso recursivo} \end{cases}$$

# Solución al ejemplo 2

## Algoritmo 2 (Recursivo)

```cpp
#include <bits/stdc++.h>
using namespace std;

int fact(int n){
    if(n == 0)
        return 1;
    return fact(n - 1) * n;
}

int main(){
    int n = 3;
    cout << fact(n) << '\n';
    return 0;
}
```

**Complejidad:** $O(n)$

# Ejemplo 3

## Fibonacci

**Descripción**

Dado un número $n$ debe encontrar el $n$.- ésimo término de la serie fibonacci

los primeros números de la serie fibonacci son:

$0, 1, 1, 2, 3, 5, 8, 13, 21, ...$

**Entrada**

La entrada consiste de un número entero $n$ $(1 \leq n \leq 10)$.

**Salida**

La salida consiste de una línea con el $n$ - ésimo número fibonacci

| Ejemplo Entrada | Ejemplo Salida |
|---|---|
| 7 | 13 |

# Análisis de la solución al ejemplo 2

**Caso Base**

El primer termino de la serie fibonacci es $0$ y el segundo es $1$, entonces:

- cuando $n = 0$, return 0
- cuando $n = 1$, return 1

**Caso Recursivo**

la serie fibonacci esta definido matemáticamente como $f(n) = f(n-1) + f(n-2)$
Expresado de otra manera:

$$fibo(n) = \begin{cases} 0 & \text{si } n = 0 \text{ caso base} \\ 1 & \text{si } n = 1 \text{ caso base} \\ fibo(n-1) + fibo(n-2) & \text{caso recursivo} \end{cases}$$

# Solución al ejemplo 3

## Algoritmo 3 (Recursivo)

```cpp
#include <bits/stdc++.h>
using namespace std;

int fibo(int n){
    if(n == 0)
        return 0;

    if(n == 1)
        return 1;

    return fibo(n - 1) + fibo(n - 2);
}

int main(){

    int n = 6;
    cout << fibo(n) << '\n';
    return 0;
}
```

**Complejidad:** $O(2^n)$

# Solución al ejemplo 3

## Algoritmo 3.1 (Recursivo)

```cpp
#include <bits/stdc++.h>
using namespace std;

int fibo(int n){
    if(n <= 1)
        return n;
   return fibo(n - 1) + fibo(n - 2);
}

int main(){

    int n = 6;
    cout << fibo(n) << '\n';
    return 0;
}
```

**Complejidad:** $O(2^n)$

# 3

# Exponenciación Rápida

# Calcular potencias

## Potencias

**Descripción**

Un problema clásico es hallar el valor de $u$, donde $u = b^e$

**Entrada**

La entrada consiste de dos números enteros $b$ y $e$, $(1 \leq b, e \leq 10^5)$, donde $b$ es la base y $e$ es el exponente.

**Salida**

Mostrar el valor de $u$

| Ejemplo Entrada | Ejemplo Salida |
|---|---|
| 2 17 | 131072 |

# Análisis de la solución al ejemplo

Como $b^e$ es la multiplicación de la base $b$, $e$ veces, es decir:

$$u = \underbrace{b \cdot b \cdot b \dots b}_{e-veces}$$

ahora como el valor máximo de $e$ es $10^5$, el algoritmo trivial que revuelve dicho problema es el siguiente:

**Algoritmo 4 Complejidad:** $O(e)$

```
#include <bits/stdc++.h>
using namespace std;

int main(){

    int b = 2, e = 17;
    int u = 1;
    for(int i = 1; i <= e; i++){
        u = u * b;
    }
    return 0;
}
```

# Calcular potencias 2

## Potencias

**Descripción**

Un problema clásico es hallar el valor de $u$, donde $u = b^e$

**Entrada**

La entrada consiste de dos números enteros $b$ y $e$, $(1 \leq b, e \leq 10^9)$, donde $b$ es la base y $e$ es el exponente.

**Salida**

Mostrar el valor de $u$

| Ejemplo Entrada | Ejemplo Salida |
|---|---|
| 2 17 | 131072 |

# Análisis de la solución al ejemplo

Como $b^e$ es la multiplicación de la base $b$, $e$ veces, es decir:

$$u = \underbrace{b \cdot b \cdot b \dots b}_{e-veces}$$

ahora como el valor máximo de $e$ es $10^9$, no se puede hacer el **algoritmo 4**.

# Solución Eficiente

Con un poco de matemáticas se puede encontrar una mejor solución

$$
\begin{aligned}
u &= b^{e} \\
u &= b^{(e \cdot \frac{2}{2})} \\
u &= b^{(2 \cdot \frac{e}{2})} \\
u &= \left(b^{2}\right)^{\frac{e}{2}}
\end{aligned}
$$

así podemos calcular $u$ dividiendo en $2$ el exponente

# Solución Eficiente

con la ecuación obtenida anteriormente, si el exponente es $par$, no hay problema porque no hay residuo, pero si es impar, se pierde información

**Ejemplo**

$b = 2$, $e = 3$, $b^e = 2^3 = 8$, pero utilizando la ecuación anterior, el resultado es $4$.

Cuando se divide un número impar entre $2$, el residuo siempre es $1$, entonces estariamos perdiendo un valor de $b$, para evitar ese caso, cada vez que $e$ es impar, se multiplica el valor de $b$ a la respuesta

# Solución Eficiente

Finalmente podemos plantear la siguiente función recursiva

**Caso Base**

Matematicamente si $e = 0$, entonces $b^0 = 1$, por tanto.

- cuando $e = 0$, return 1

**Caso Recursivo**

tiene dos partes:

- cuando $e$ es par, entonces $potencia(b^2, \frac{e}{2})$
- cuando $e$ es impar, entonces $potencia(b^2, \frac{e}{2}) \cdot b$

Expresado de otra manera:

$$potencia(b, e) = \begin{cases} 1 & \text{si } e = 0 \text{ caso base} \\ potencia(b^2, \frac{e}{2}) & \text{si } e \text{ es par} \\ potencia(b^2, \frac{e}{2}) \cdot b & \text{si } e \text{ es impar} \end{cases}$$

# Solución eficiente

## Algoritmo 4.1

```cpp
#include <bits/stdc++.h>
using namespace std;

int potencia(int b, int e){
    if(e == 0)
        return 1;

    if(e % 2 == 0)
       return potencia(b * b, e / 2);
   return potencia(b * b, e / 2) * b;


}

int main(){

    int b = 2, e = 17;
    cout << potencia(b, e) << '\n';
    return 0;
}
```

**Complejidad:** $O(\log(e))$

# Versión iterativa

## Algoritmo 4.2

```cpp
#include <bits/stdc++.h>
using namespace std;

int potencia(int b, int e){
  int u = 1;
  while(e){
    if(e & 1)
      u = u * b;
    e /= 2;
    b = b * b;
  }
  return u;
}

int main(){

    int b = 2, e = 17;
    cout << potencia(b, e) << '\n';
    return 0;
}
```

**Complejidad:** $O(\log(e))$

# Complejidad

Dado que la potencia $e$ es la que determina cuantas veces debe entrar a la recursión, entonces este parámetro es la que determina la complejidad de la funcion **potencia**
Ahora, $e$ siempre se va dividiendo en $2$, al igual que la búsqueda binaria, la complejiddad del algoritmo es $O(\log e)$

# 4

# Exponenciación Matricial

# Matrix

**Descripción**

Un problema clásico es hallar el valor de $u$, donde $u = X^e$ donde $X$ es una matriz cuadrada

**Entrada**

La entrada consiste de varias lineas, la primera linea tiene un número entero $n$, $e$ $(1 \leq n \leq 10)$, $(1 \leq e \leq 10^5)$ donde $n$ es el tamaño de la matriz y $e$ es la potencia de $X$.

Luego vienen $n$ lineas, cada linea tiene $n$ números $(1 \leq A_{i,j} \leq 10^5)$, los elementos de la matriz.

**Salida**

Mostrar el valor de $u$, imprimir todo módulo $10^9 + 7$.

| Ejemplo Entrada | Ejemplo Salida |
|---|---|
| 2 2 | 7 10 |
| 1 2 | 15 22 |
| 3 4 | |

# Análisis de la solución al ejemplo

Como $X^e$ es la multiplicación de la matriz $X$, $e$ veces, es decir:

$$u = \underbrace{X \cdot X \cdot X \ldots X}_{e-veces}$$

ahora como el valor máximo de $e$ es $10^5$, el algoritmo trivial que revuelve dicho problema es el siguiente:

**Algoritmo:** **Complejidad:** $O(n^3 \cdot e)$

# Solución

```
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
ll const mod = 1e9 + 7;
vector<vector<ll>> mul_matrix(vector<vector<ll>> &A, vector<vector<ll>> &B){
    int n = (int) A.size();
    vector<vector<ll>> ans(n, vector<ll> (n));
     for(int i = 0; i < n; i++){
       for(int j = 0; j < n; j++){
           for(int k = 0; k < n; k++){
               ans[i][j] = (ans[i][j] + A[i][k] * B[k][j]) % mod;
           }
       }
    }
    return ans;
}
int main(){
    vector<vector<ll>> X = {{1, 2}, {3, 4}};
    vector<vector<ll>> ans = {{1, 0}, {0, 1}}; // matriz identidad
    int e = 2;
    for(int i = 0; i < e; i++){
        ans = mul_matrix(ans, X); // en ans esta la respuesta
    }
    return 0;
}
```

# Matrix 2

**Descripción**

Un problema clásico es hallar el valor de $u$, donde $u = X^e$ donde $X$ es una matriz cuadrada

**Entrada**

La entrada consiste de varias lineas, la primera linea tiene un número entero $n$, $e$ $(1 \leq n \leq 10)$, $(1 \leq e \leq 10^9)$ donde $n$ es el tamaño de la matriz y $e$ es la potencia de $X$.

Luego vienen $n$ lineas, cada linea tiene $n$ números $(1 \leq A_{i,j} \leq 10^5)$, los elementos de la matriz.

**Salida**

Mostrar el valor de $u$, imprimir todo módulo $10^9 + 7$.

| Ejemplo Entrada | Ejemplo Salida |
|---|---|
| 2 2<br>1 2<br>3 4 | 7 10<br>15 22 |

# Análisis de la solución al ejemplo

Como $X^e$ es la multiplicación de la base $X$, $e$ veces, es decir:

$$u = \underbrace{X \cdot X \cdot X \ldots X}_{e-veces}$$

ahora como el valor máximo de $e$ es $10^9$, no se puede hacer el **anterior algoritmo**.

# Solución Eficiente

Con un poco de matemáticas se puede encontrar una mejor solución

$$
\begin{aligned}
u &= X^{e} \\
u &= X^{(e \cdot \frac{2}{2})} \\
u &= X^{(2 \cdot \frac{e}{2})} \\
u &= (X^{2})^{\frac{e}{2}}
\end{aligned}
$$

así podemos calcular $u$ dividiendo en $2$ el exponente

# Solución Eficiente

con la ecuación obtenida anteriormente, si el exponente es $par$, no hay problema porque no hay residuo, pero si es impar, se pierde información

**Ejemplo**

Cuando se divide un número impar entre $2$, el residuo siempre es $1$, entonces estariamos perdiendo un valor de $X$, para evitar ese caso, cada vez que $e$ es impar, se multiplica la matriz $X$ a la respuesta

# Solución Eficiente

Finalmente podemos plantear la siguiente función recursiva

**Caso Base**

Matematicamente si $e = 0$, entonces $X$, es la matriz identidad.

- cuando $e = 0$, return matrix identidad

**Caso Recursivo**

tiene dos partes:

- cuando $e$ es par, entonces $power(X^2, \frac{e}{2})$
- cuando $e$ es impar, entonces $power(X^2, \frac{e}{2}) \cdot X$

Expresado de otra manera:

$$power(X, e) = \begin{cases} matriz \ \ identidad & \text{si } e = 0 \text{ caso base} \\ power(X^2, \frac{e}{2}) & \text{si } e \text{ es par} \\ power(X^2, \frac{e}{2}) \cdot X & \text{si } e \text{ es impar} \end{cases}$$

# Solución eficiente

```cpp
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
int const N = 4e3 + 100;
ll const mod = 1e9 + 7;
vector<vector<ll>> power(vector<vector<ll>> A, ll e){
    if(e == 0){
        vector<vector<ll>> ans(A.size(), vector<ll> (A.size()));
        for(int i = 0; i < A.size(); i++)
            ans[i][i] = 1;
        return ans;
    }
    vector<vector<ll>> ans = power(mul_matrix(A, A), e / 2);
    if(e % 2 == 1)
        ans = mul_matrix(ans, A);
    return ans;
}
int main(){
    vector<vector<ll>> X = {{1, 2}, {3, 4}}, ans = power(X, 2);
    // en ans esta la respuesta
    return 0;
}
```

**Complejidad:** $O(n^3 \cdot \log(e))$

# Algoritmo Versión iterativa

**Complejidad:** $O(n^3 \cdot \log(e))$

```
#include <bits/stdc++.h> // java.util.*;
using namespace std;
typedef long long ll;
int const N = 4e3 + 100;
ll const mod = 1e9 + 7;
 vector<vector<ll>> power(vector<vector<ll>> A, ll b){
    int n = (int) A.size();
    vector<vector<ll>> ans(n, vector<ll> (n));
    for(int i = 0; i < n; i++)
        ans[i][i] = 1;
    while(b){
        if(b & 1)
            ans = mul_matrix(ans, A);
        b >>= 1;
        A = mul_matrix(A, A);
    }
    return ans;
}
int main(){
    vector<vector<ll>> X = {{1, 2}, {3, 4}}, ans = power(X, 2);
    // ans esta la respuesta
    return 0;
}
```

# Complejidad

Dado que la potencia $e$ es la que determina cuantas veces debe entrar a la recursión, entonces este parámetro es la que determina la complejidad de la funcion **potencia**, además hay que multiplicaar por el costo que multiplicar matrices que en este caso es cúbico $n^3$
Ahora, $e$ siempre se va dividiendo en $2$, al igual que la búsqueda binaria, la complejiddad del algoritmo es $O(n^3 \cdot \log e)$

¿Preguntas?

¡Gracias!