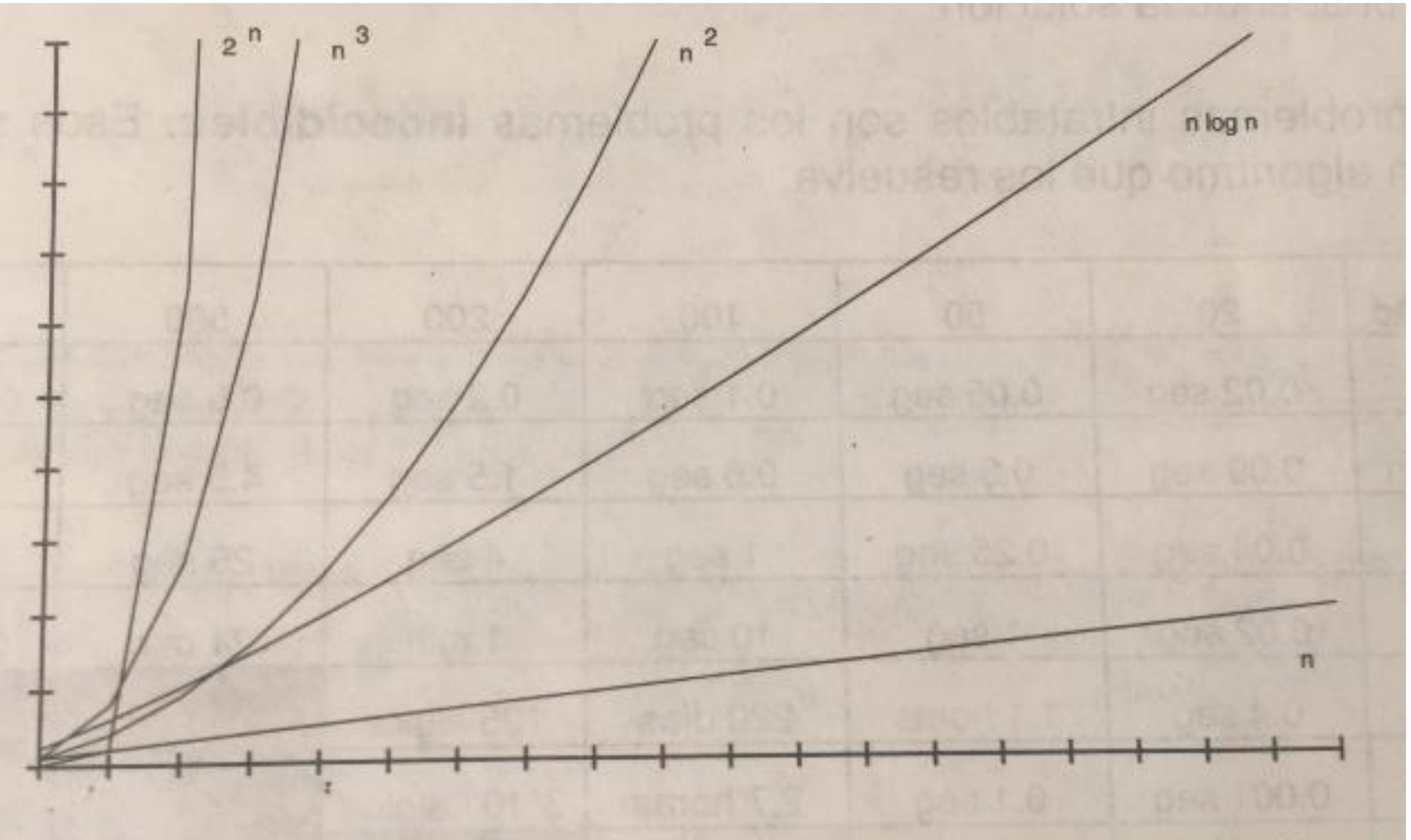
COMPLEJIDAD COMPUTACIONAL Notación O

Carlos Augusto Meneses E. (2018)

Tomado de "Diseño y Estructuras de Datos en C" – Jorge Villalobos

1. El concepto de Complejidad Computacional

La idea es tratar de encontrar una función **f(n)**, fácil de calcular que acote el crecimiento de la función de tiempo → "**T_A(n)** crece aproximadamente como **f**"



Crecimiento de las funciones típicas de complejidad de algoritmos

1. Complejidad Computacional – Problemas

Ejemplo: Para los 8 algoritmos A1,…,A8, supongamos que procesar 1 dato, tarde 1 microsegundo → Veamos el tamaño max del problema a resolver:

	Complejidad	1 seg	10^2 seg (1.7 min)	10^4 seg (2.7 horas)	10^6 seg (12 días)	10^8 seg (3 años)	10^{10} seg (3 siglos)
A1	$1000n$	10^3	10^5	10^7	10^9	10^{11}	10^{13}
A2	$1000n\log_2 n$	$1.4 * 10^2$	$7.7 * 10^3$	$5.2 * 10^5$	$3.9 * 10^7$	$3.1 * 10^9$	$2.6 * 10^{11}$
A3	$100n^2$	10^2	10^3	10^4	10^5	10^6	10^7
A4	$10n^3$	46	$2.1 * 10^2$	10^3	$4.6 * 10^3$	$2.1 * 10^4$	10^5
A5	$n^{\log 2n}$	22	36	54	79	112	156
A6	$2^{n/3}$	59	79	99	119	139	159
A7	2^n	19	26	33	39	46	53
A8	3^n	12	16	20	25	29	33

1. Complejidad Computacional – Problemas

Ejemplo: Un algoritmo de complejidad $O(2^n)$ resuelve un problema de tamaño 20 en 1 seg, pero con tamaño 50 ya no es viable (35 años) buscando la solución.

Complejidad	20	50	100	200	500	1000
$1000n$	0.02 seg	0.05 seg	0.1 seg	0.2 seg	0.5 seg	1 seg
$1000n\log_2 n$	0.09 seg	0.3 seg	0.6 seg	1.5 seg	4.5 seg	10 seg
$100n^2$	0.04 seg	0.25 seg	1 seg	4 seg	25 seg	2 min
$10n^3$	0.02 seg	1 seg	10 seg	1 min	21 min	2.7 horas
n^{log2n}	0.4 seg	1.1 horas	220 días	125 siglos		
$2^{n/3}$	0.001 seg	0.1 seg	2.7 horas	$3*10^4$ siglos		
2^n	1 seg	35 años	$3*10^4$ siglos			
3^n	58 min	$2*10^9$ siglos				

Estimativos de tiempo para resolver un problema de tamaño

1. Complejidad Computacional – Notación O

"El algoritmo es **O**(***f***(**n**))": Quiere decir que al aumentar el número de datos que debe procesar, el tiempo del algoritmo va a crecer, como crece ***f*** en relación a **n**.

Definición de Complejidad O:

$T_A(n)$ es $O(f(n))$ (la complejidad de A es $f(n)$) <u>ssi</u> $\exists\, c, n_0 > 0 \mid \forall\, n \geq n_0,\ T_A(n) \leq c\, f(n)$

1. Complejidad Computacional – Teorema 1

Si $T_A(n)$ es $O(k\,f(n)) \Rightarrow T_A(n)$ también es $O(f(n))$.

Este teorema expresa una de las bases del análisis de algoritmos: lo importante no es el valor exacto de la función que acota el tiempo, sino su forma. Esto permite eliminar todos los factores constantes de la función cota. Por ejemplo, un algoritmo que es $O(2n)$ también es $O(n)$, puesto que ambas funciones tienen la misma forma, aunque tienen diferente pendiente.

<u>Demostración:</u>

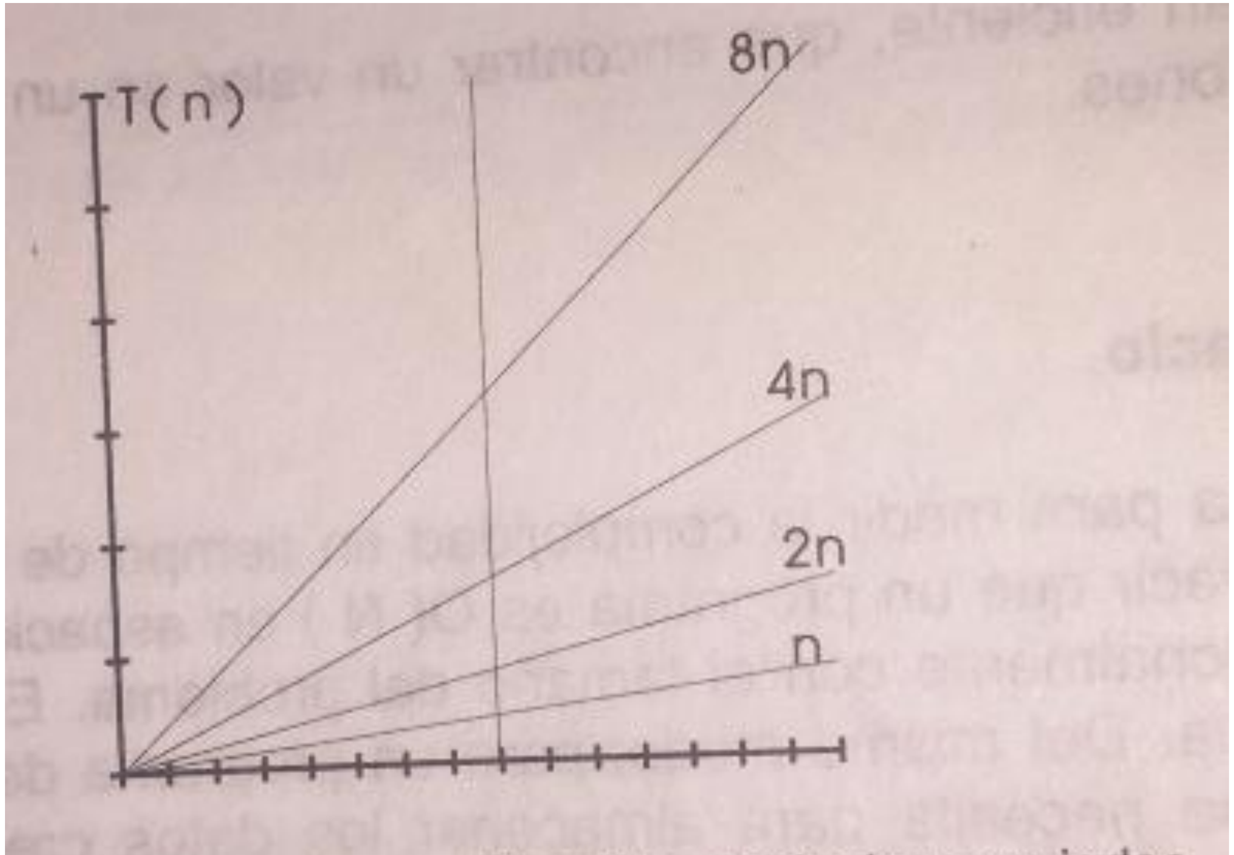
Si $T_A(n)$ es $O(k\,f(n)) \Rightarrow$ $\exists\, c, n_0 > 0 \mid \forall\, n \geq n_0,\ T_A(n) \leq c.k.f(n)$

Al tomar $c_1 = c.k > 0$ se tiene que

$\exists\, c_1, n_0 > 0 \mid \forall\, n \geq n_0\ T_A(n) \leq c_1.f(n) \Rightarrow$

$T_A(n)$ es $O(f(n))$ ◆

1. Complejidad Computacional – Teorema 1



Funciones cota con diferentes constantes asociadas

1. Complejidad Computacional – Teorema 2

Si A1 y A2 son algoritmos, tales que $T_{A1}(n)$ es $O(f1(n))$ y $T_{A2}(n)$ es $O(f2(n))$, el tiempo empleado en ejecutarse A1 seguido de A2 es $O(\max(f1(n), f2(n)))$.

Esto quiere decir que si se tienen dos bloques de código y se ejecuta uno después del otro, la complejidad del programa resultante es igual a la complejidad del bloque más costoso. Por esta razón, si hay una secuencia de comandos $O(1)$, también esta secuencia tendrá, en conjunto, complejidad constante. Pero si alguna de sus instrucciones es $O(n)$, todo el programa será $O(n)$.

1. Complejidad Computacional – Teorema 2

Demostración:

Si $T_{A1}(n)$ es $O(f1(n)) \Rightarrow \quad \exists\, c_1, n_1 > 0 \mid \forall\, n \geq n_1,\ T_{A1}(n) \leq c_1.f_1(n)$

Si $T_{A2}(n)$ es $O(f2(n)) \Rightarrow \quad \exists\, c_2, n_2 > 0 \mid \forall\, n \geq n_2,\ T_{A2}(n) \leq c_2.f_2(n)$

$\Rightarrow T_A(n) = T_{A1}(n) + T_{A2}(n) \leq c_1.f_1(n) + c_2.f_2(n),\ \forall\, n \geq \max(n1, n2) \Rightarrow$

$\Rightarrow T_A(n) \leq (c_1 + c_2) * \max(f_1(n), f_2(n)),\ \forall\, n \geq \max(n1, n2) \Rightarrow$

Al tomar: $\quad n_0 = \max(n_1, n_2) > 0$

$c_0 = (c_1 + c_2) > 0$, se tiene que:

$\exists\, c_0, n_0 > 0 \mid \forall\, n \geq n_0\ T_A(n) \leq c_0.\max(f_1(n), f_2(n))$

$\Rightarrow T_A(n)$ es $O(\max(f1(n), f2(n)))$ ◆

1. Complejidad Computacional – Teorema 3

Sea A1 un algoritmo que se repite itera(n) veces dentro de un ciclo, tal que itera(n) es $O(f2(n))$ y $T_{A1}(n)$ es $O(f1(n))$. El tiempo de ejecución del programa completo $T_A(n) = T_{A1}(n) * itera(n)$ es $O(f1(n) * f2(n))$ suponiendo que el tiempo de evaluación de la condición se encuentra incluido en el tiempo de ejecución de algoritmo A1.

1. Complejidad Computacional – Teorema 3

<u>Demostración:</u>

Si $T_{A1}(n)$ es $O(f1(n)) \Rightarrow \quad \exists\, c_1, n_1 > 0 \mid \forall\, n \geq n_1 \; T_{A1}(n) \leq c_1.f_1(n)$

Si $itera(n)$ es $O(f2(n)) \Rightarrow \quad \exists\, c_2, n_2 > 0 \mid \forall\, n \geq n_2 \; itera(n) \leq c_2.f_2(n)$

$\Rightarrow T_A(n) = T_{A1}(n) * itera(n) \leq c_1.f_1(n) * c_2.f_2(n), \; \forall\, n \geq max(n1, n2) \Rightarrow$

$\Rightarrow T_A(n) \leq (c_1 * c_2) * f_1(n) * f_2(n) \Rightarrow$

Al tomar: $\quad n_0 = max(n_1, n_2) > 0$

$c_0 = (c_1 * c_2) > 0$, se tiene que:

$\exists\, c_0, n_0 > 0 \mid \forall\, n \geq n_0 \; T_A(n) \leq c_0.f_1(n).f_2(n)$

$\Rightarrow T_A(n)$ es $O(f1(n) * f2(n))$ ◆

2. Complejidad en el espacio

- La idea es la misma que en la complejidad temporal. i.e. O(n) significa que sus requerimientos de memoria aumentan proporcionalmente al tamaño del problema.

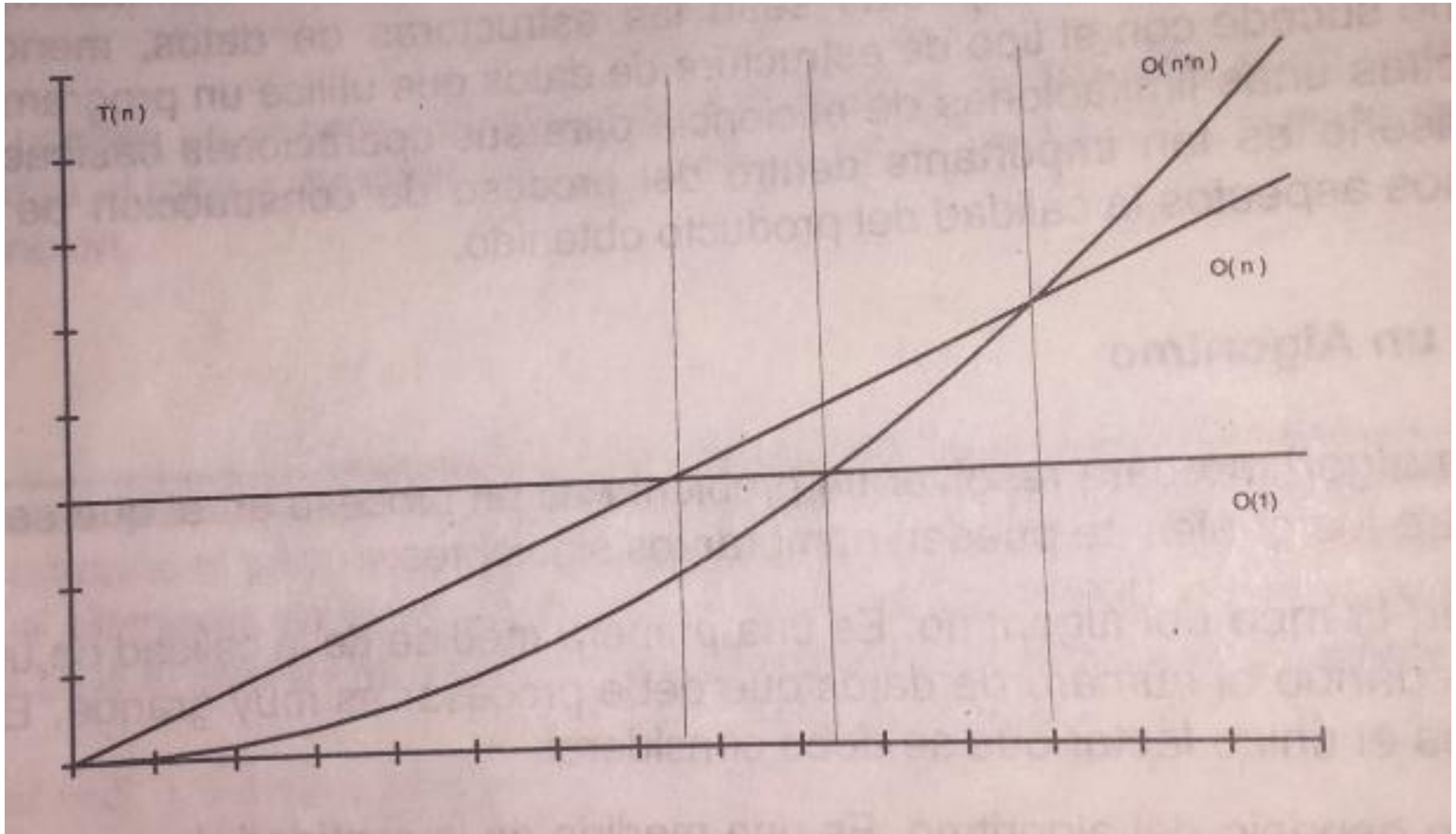
Eficiencia de un programa: Tiene que ver con el tiempo y espacio utilizado.
Las estructuras de datos deben ser adecuadas para mejorar la eficiencia.

2. Selección de un algoritmo

Se tienen en cuenta algunos factores en el algoritmo:

- La complejidad en el tiempo.
- La complejidad en el espacio.
- Dificultad de implementación.
- EL tamaño del problema. En problemas pequeños, muchas veces no importan los factores anteriores.
- El valor de la constante asociada a la función de complejidad O(k* f(n))

2. Complejidad Computacional – comparación y selección algorítmica

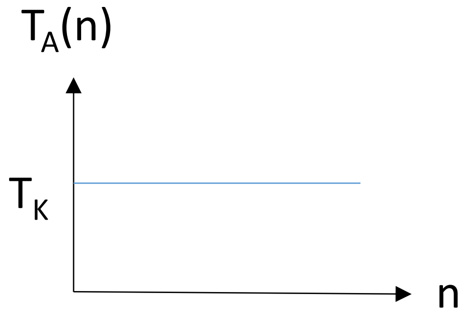


Comparación de varias funciones para valores pequeños de un problema

3. Ejercicios

1) Calcular la complejidad de la asignación:

    var = 5



R/

- T_K es el tiempo que toma la asignación
- $T_A(n)$ es $O(T_K)$
- → $T_A(n)$ es $O(1)$ <por teorema 1>

2) Calcular la complejidad de:

```
a = 1;
b = 2;
c = 3;
```

3. Ejercicios

3)

```
float abs (float n)
{ if (n < 0)
      return –n;
  else
      return n;
}
```

R/

$T_{abs}(n) <= T_{cond} + \max (T_{return} , T_{return}) <= T_{cond} + T_{return}$

$\rightarrow T_{abs}(n)$ es $O (\max (1 , 1))$

$\rightarrow O (1)$

Por lo tanto, $x = abs (y)$ es $O(1)$

3. Ejercicios

4)

```
int factorial (int n)
{ int i, acum;
   i = 0;
   acum = 1;
   while (i < n)
      {  i++;
         acum *= i;
      }
     return acum;
}
```

O(1)

while itera n veces → f(n) = n acota T_{while} → O(n)

O(1)

R/

$T_{factorial}(n)$ es O (max (1 , n, 1))

→ O (n)

Por lo tanto, x = factorial (y) es O(n)

3. Ejercicios

```
5)      for (i=0; i<5; i++)
            a[i]=i;
R/
        Es lo mismo que:
a[i]=0;
a[i]=1;
a[i]=2;
a[i]=3;
a[i]=4;
        → O (1)
```

3. Ejercicios

6)

```
void inic(int a[], int tam)
{  int i;
   for (i=0; i<tam; i++)
      a[i]=i;
```

R/

Nro de asignaciones no es fijo → O (n)

3. Ejercicios

7)

```
for (i=0; i<N; i++)          → O(N)
    for (j=0; j<M; j++)      → O(M)
        w[i][j]= i*j;        → O(1)
```

R/

→ O (N*M) → $O(n^2)$

3. Ejercicios – complejidad en recursión

8)

```
int factorial (int num)
    if (num == 0)
        return 1;
    else
        return num*factorial (num-1);
```

R/

$$T(num) = \begin{cases} T_K & \text{si } num=0 \\ T_K + T(num-1) & \text{si } num>0 \end{cases}$$

$$\begin{aligned} T(num) &= T_K + T(num-1) \\ &= T_K + T_K + T(num-2) \\ &\dots \\ &= num * T_K + T(0) \\ &= T_K * T(num+1) \rightarrow O(num) \end{aligned}$$

¿Cuál será la complejidad en el espacio?

3. Ejercicios – complejidad búsqueda binaria

9) int binaria (int v, int e, int inf, int sup)

```
{ int medio = (inf+sup+1)/2;
  if (inf > sup)
    return FALSE;
  else  if (v[medio] == e)
     return TRUE;
  else if (v[medio]>e)
    return binaria(v, e, inf, medio-1);
  else
    return binaria(v, e, medio+1, sup);
}
```

3. Ejercicios – complejidad búsqueda binaria

R/

$$T(n) = \begin{cases} 1 & \text{si } n<=1 \\ 1 + T(n/2) & \text{si } n>1 \end{cases}$$

$$\begin{aligned} T(n) &= 1 + T(n/2) \\ &= 1 + 1 + T(n/4) \\ &= 1 + 1 + T(n/8) \\ &\quad \ldots \\ &= \log_2 n * 1 + T(n/n) \\ &= \log_2 n + 1 \end{aligned}$$

$\rightarrow$ $T(n)$ es $O(\log_2 n)$

4. Especificación de un programa

Un ***programa*** es una secuencia de instrucciones que transforma un ***estado inicial*** en un ***estado final***, que resuelve un problema.

→ Un programa se puede especificar mediante 2 aserciones:

- Estado inicial: Condiciones de datos de entrada
- Estado final: Condiciones de datos de salida.

Ejemplo:
{Pre: num>=0}
{Post: fact = num!}

3. Ejercicios – Especificar y calcular complejidad

Para los siguientes ejercicios, hacer la especificación de la función con pre y post condición, implementar el algoritmo, calcular la complejidad.

1) Hacer una función que reciba un número entero positivo y retorne verdadero si es primo o falso sino.
2) Hacer una función que reciba un valor N y retorne la sumatoria de los enteros hasta N.
3) Hacer una función que llene un vector (de tamaño N) con números enteros aleatorios (puede ser de 4 cifras).
4) Hacer una función que llene una matriz de tamaño NxM con números aleatorios.
5) Hacer una función que reciba 2 matrices de tamaño NxM y MxP respectivamente y devuelva otra matriz con la multiplicación de las dos primeras

3. Ejercicios – Especificar y calcular complejidad

6) Hacer una función que calcule el n-ésimo número de la serie de fibonacci,
7) Hacer una función que retorne el mcm (mínimo común múltiplo) de 2 numeros naturales mayores que 0.
8) Hacer una función que calcule el número de combinaciones , en una expresión de combinación binomial . Conocemos el valor de k y n (n> 0, k>=0 y k<=n). Co ibles combinaciones para seleccionar k elementos de u ementos. Esta expresión está dada por la fórmula siguie

$$\binom{n}{k} = \frac{n!}{k!\,(n-k)!}$$