> Reinventa el mundo <

CLASE 1: INTRODUCCIÓN

CS8077- Estructura de datos y algoritmos

Ciclo 2025

Heider Sanchez
hsanchez@utec.edu.pe

TRANSFORMATEC

Qué veremos en este curso...

- ❑Algoritmos para resolver problemas de manera eficiente
- ❑Estructuras de datos para almacenar y organizar datos eficientemente
- ❑Análisis de requerimientos para uso adecuado de distintas estructuras de datos

Qué NO veremos en este curso...

- POO se asume como algo estudiado en cursos previos.

- No se enseñará a compilar en C++, ni a usar algún IDE específico.

- El manejo de Punteros en C++ se asume como algo aprendido en un nivel intermedio-avanzado.

- La STL y estructuras básicas se entienden como ya vistas en cursos previos.
  Tutorial rápido de STL C++

Lista de temas

Semana	Tema
1	Complejidad, Complejidad
2	Array, Listas simples
3	Listas dobles
4	Pilas y Colas,
5	Tabla Hash,
6	Árbol Binario de Búsqueda,
7	Árboles AVL
8	Heaps, Disjoint Set
9	Árboles B / B+
10	Tries, stringmatching
11	Grafos
12	Búsqueda en Grafos 1
13	Búsqueda en Grafos 2
14	Matrices Dispersas
15	Optimización: Backtracking, Hill Climbing, (NP Hard, Analisis Combinatorio)

Laboratorio

1. Si el entregable no compila será calificado sobre 11
2. Tratar de evitar warnings en los proyectos
3. Tratar que pasen todos los tests de prueba.
4. No olviden subir su código a master

UTEC
UNIVERSIDAD DE INGENIERÍA Y TECNOLOGÍA

Sistema de evaluación

	TEORÍA (T)	LABORATORIO (L)
EVALUACIÓN * La ponderación de la evaluación se hará si ambas partes están aprobadas	Examen **E1** (20%) Examen **E2** (20%)	Evaluación Continua **C1** (20%) Evaluación Continua **C2** (20%) Proyecto **P1** (10%) Proyecto **P2** (10%)
	40%	60%
	100%	

Evaluación Continua:

- Ejercicios individuales resueltos en clase
- Tareas grupales en Github Classroom

Herramientas

- Compilador: C++ 17

  (GCC, Cygwin, MinGW-w64, MSYS2)

- IDE: Visual Code, CLion

- LeetCode

- Karma Git Commits
  a. *http://karma-runner.github.io/4.0/dev/git-commit-msg.html*

Revisar Git / GitHub

1. Crear una cuenta en github (https://github.com/)
2. Revisar: clone/add/commit/push/pull
3. Hacer push de un main.cpp (Hello world!)
4. Revisar branch/checkout/merge
5. Crear una rama develop, modificar el main.cpp
6. Hacer un commit y merge con master

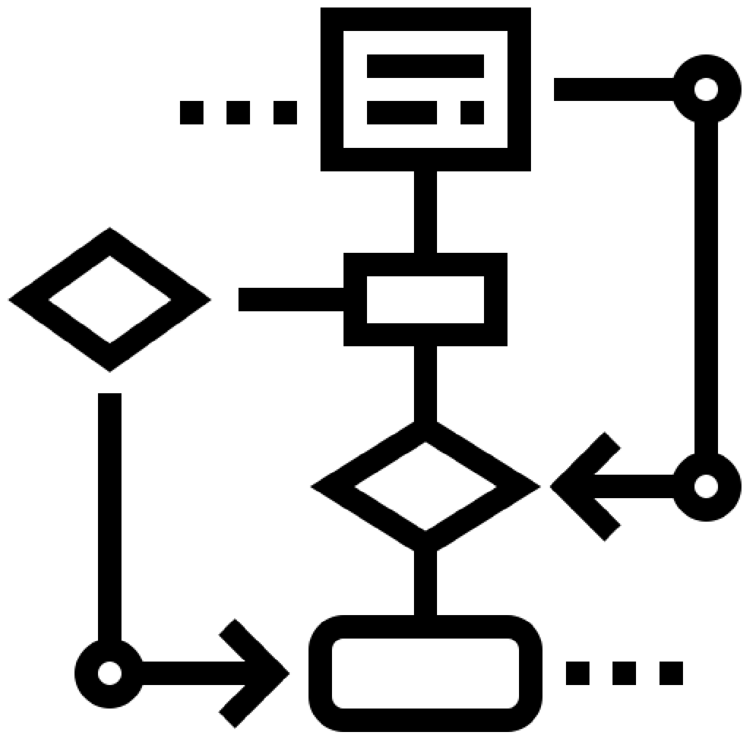
1.

Complejidad Algorítmica

¿Qué es un algoritmo?

- Es un conjunto ordenado de pasos o instrucciones, los cuales son realizados para resolver un problema.

- Un algoritmo siempre debe producir un resultado, por ello se debe hacer de forma racional y con un objetivo en mente.



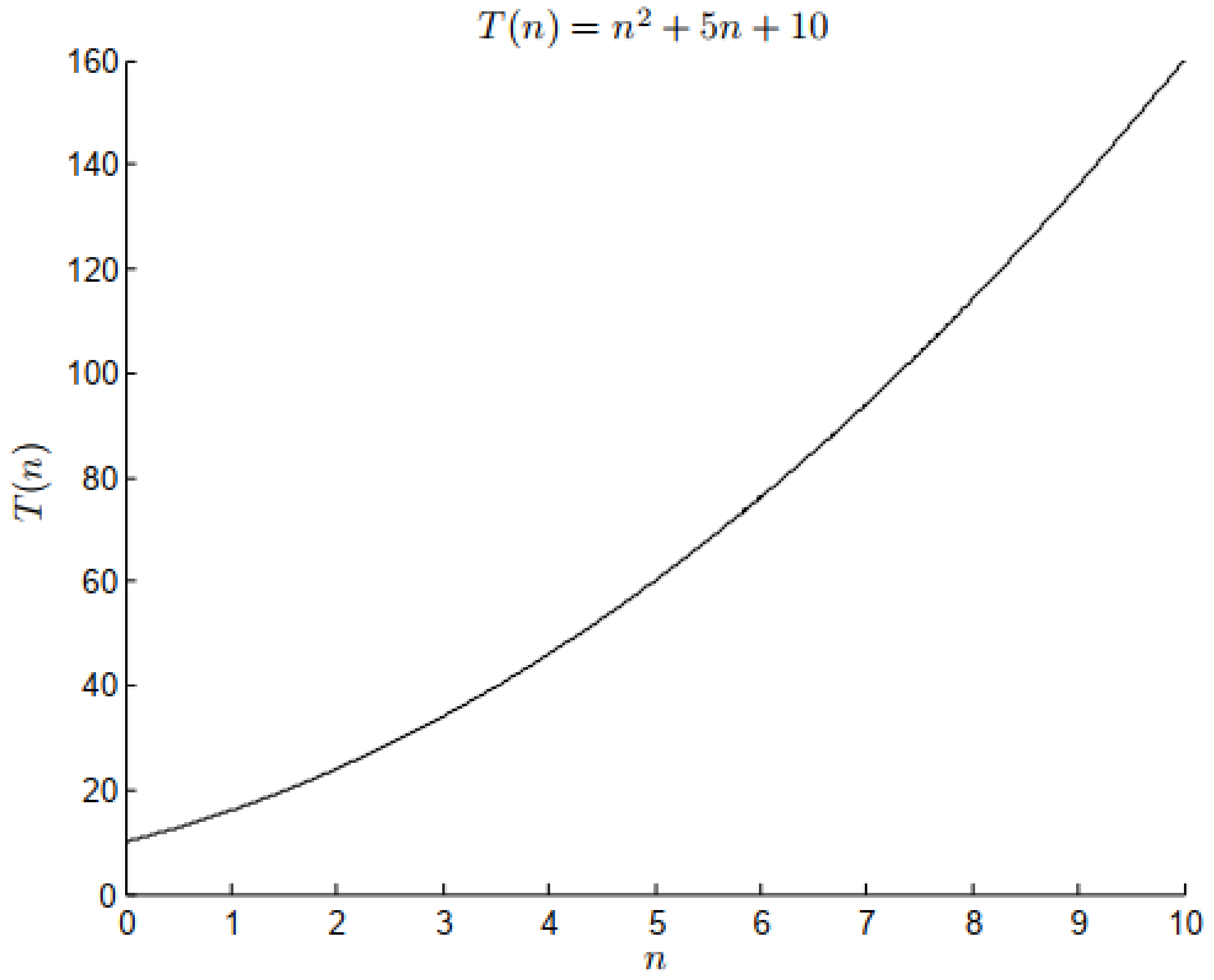
E.g. Algoritmos de búsqueda, algoritmos de ordenamiento, algoritmos criptográficos. Etc.

Análisis de Algoritmos

- Para **cuantificar la eficiencia** de los algoritmos, utilizamos funciones que miden, por ejemplo:
  - Cuánto **tiempo demora** un algoritmo en ejecutarse sobre una entrada dada,
  - Cuál es su **peor caso** sobre un conjunto de entradas posibles,
    - O cuánto demora en promedio, suponiendo una cierta distribución de probabilidad de las entradas.
  - También estudiaremos el uso de otro tipo de recursos, como por ejemplo la **cantidad de memoria utilizada**.

Tiempo Computacional $T(n)$

- Cuenta el número de operaciones fundamentales que el algoritmo realiza, lo cual puede variar según el tamaño de la entrada n.
- El tiempo de ejecución lo deberemos expresar mediante una fórmula (función) matemática.



Tiempo Computacional $T(n)$

$$\sum_{i=1}^{n} i$$

```c
int suma = 0;                        // 1
for (int i = 1; i <= n; i++)         // 2n + 2
    suma += suma;                    // 2n
```

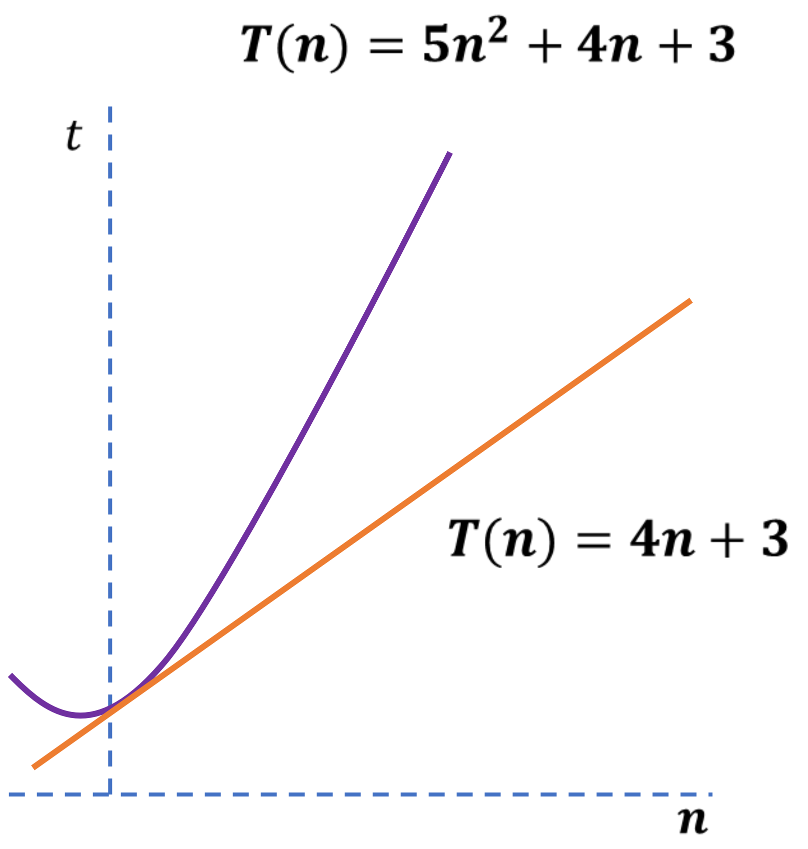
`int suma = 0;`	1
`for (int i = 1; i <= n; i++)`	$2n+2$
`    suma += suma;`	$2n$

$$\boldsymbol{T(n) = 4n + 3}$$

$$\sum_{i=1}^{n}\sum_{j=1}^{n} i * j$$

`int suma = 0;`	1
`for (int i = 1; i <= n; i++)`	$2n+2$
`    for (int j = 1; j <= n; j++)`	$(2n+2)*n$
`        suma = suma + i*j;`	$3n^2$

$$\boldsymbol{T(n) = 5n^2 + 4n + 3}$$



Tiempo Computacional $T(n)$

```
int i=1;
while(i<=n) {
    i = i * 2;
}
```

Valores de i en cada iteración x:

$2^1, 2^2, 2^3, 2^4, 2^5, \ldots, 2^x$

En algún momento:

$i > n$, es decir $2^x > n$

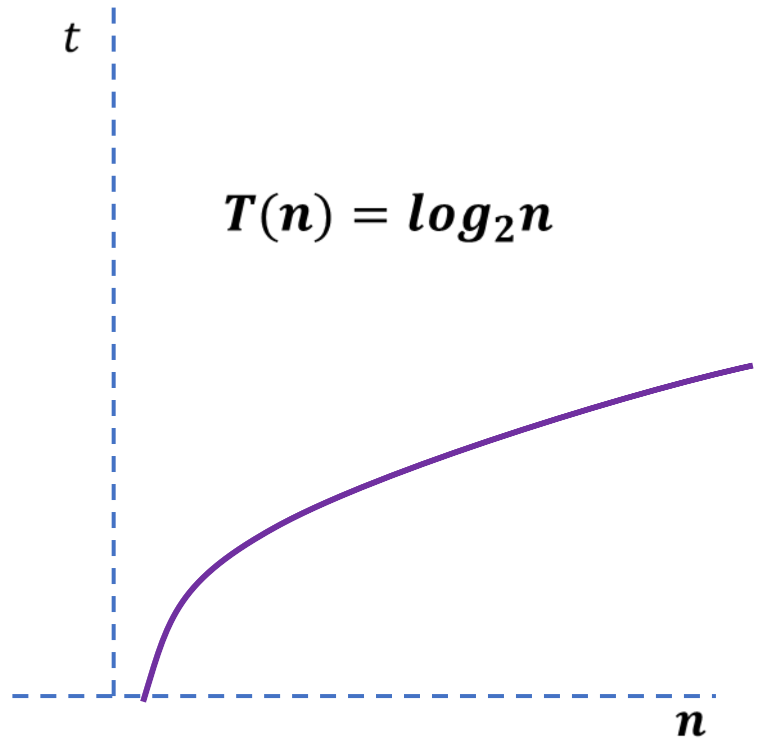
Entonces:

$2^x > n \rightarrow$

$log_2 2^x > log_2 n \rightarrow$

$x\, \cancel{log_2 2} > log_2 n \rightarrow$

$\boxed{\boldsymbol{x > log_2 n}}$ ← **Tiempo Computacional**



Tiempo Computacional $T(n)$

Análisis del Algoritmo Fibonacci Recursivo

$$fib(n) = \begin{cases} 1 & , n \leq 1 \\ fib(n-1) + fib(n-2), & n > 1 \end{cases}$$

$$T(n) = T(n-1) + T(n+2) + c$$

::

$T(n-2)$ es casi $T(n-1)$, entonces:

$$T(n) = 2T(n-1) + c$$

$$T(n) = 2\{2T(n-2) + c\} + c = 4T(n-2) + 3c$$

$$T(n) = 4\{2T(n-3) + c\} + 3c = 8T(n-3) + 7c$$

$$T(n) = 16T(n-4) + 15c$$

$$T(n) = 2^k T(n-k) + (2^k - 1)c$$

Condición de parada $T(0), n - k = 0 \rightarrow k = n$

$$T(n) = 2^n T(0) + (2^n - 1)c$$

Tiempo Computacional $\longrightarrow$ $\boxed{\boldsymbol{T(n) = (1+c)2^n - c \approx 2^n}}$

Tiempo Computacional $T(n)$

Ejercicio: hallar el tiempo computacional del siguiente algoritmo (acotado al peor caso)

```
void insertion_sort(T *array, int n)
{
    int actual, j;
    for (int i = 1; i < n; ++i)
    {
        actual = array[i];
        j = i - 1;
        while (j >= 0 && array[j] > actual)
        {
            array[j + 1] = array[j];
            --j;
        }
        array[j + 1] = actual;
    }
}
```

Tiempo Computacional $T(n)$

Ejercicio: hallar el tiempo computacional del siguiente algoritmo (acotado al peor caso)

```
void insertion_sort(T *array, int n)
{
    int actual, j;                                  // 1
    for (int i = 1; i < n; ++i)                     // n
    {
        actual = array[i];                          // n
        j = i - 1;
        while (j >= 0 && array[j] > actual)         // n^2
        {
            array[j + 1] = array[j];                // n^2
            --j;                                    // n^2
        }
        array[j + 1] = actual;                      // n
    }
}
```

$$\boldsymbol{T(n) = 3n^2 + 4n + 1}$$

Tiempo Computacional $T(n)$

Ejercicio: hallar el tiempo computacional de las siguientes sentencias

```
for (int i = 1; i <= n; i *= c ) {
    // cualquier sentencia constante
}
```

```
for (int i = n; i > 0; i /= c) {
    // cualquier sentencia constante
}
```

```
for (int i = 2; i <= n; i = pow(i, c)) {
    // cualquier sentencia constante
}
```

Tiempo Computacional $T(n)$

Ejercicio: hallar el tiempo computacional de las siguientes sentencias

```
for (int i = 1; i <= n; i *= c ) {
    // cualquier sentencia constante
}
```

$$\boldsymbol{T(n) \approx log_c n}$$

```
for (int i = n; i > 0; i /= c) {
    // cualquier sentencia constante
}
```

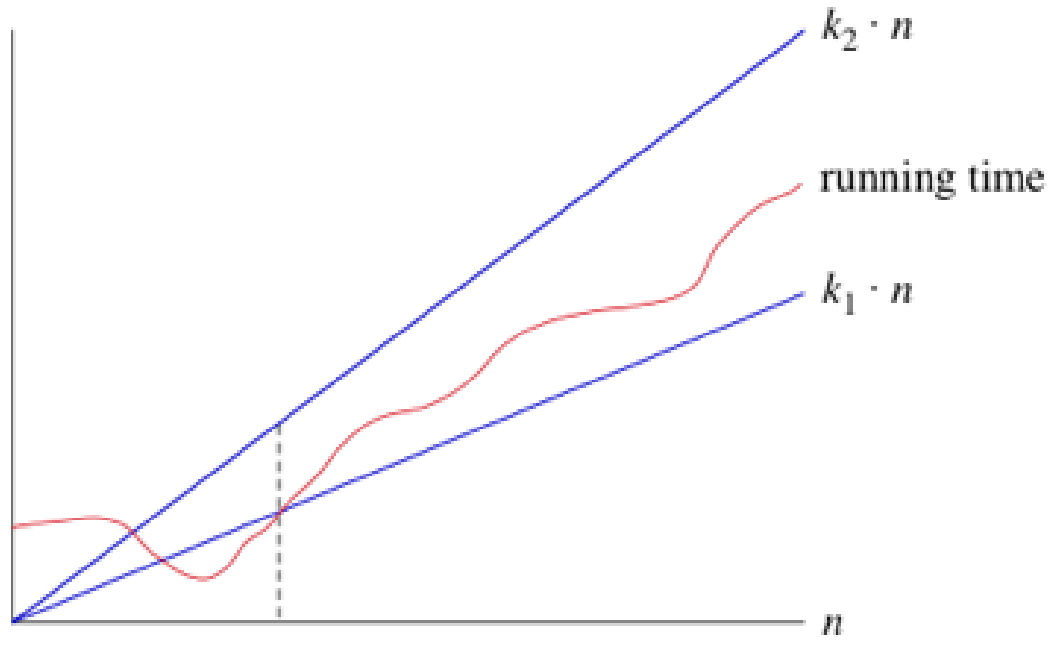
$$\boldsymbol{T(n) \approx log_c n}$$

```
for (int i = 2; i <= n; i = pow(i, c)) {
    // cualquier sentencia constante
}
```

$$\boldsymbol{T(n) \approx log\ log_c n}$$

Notación Asintótica

- Permite simplificar la taza de crecimiento de un algoritmo
- Eficiencia asintótica de algoritmos
  - Asumimos que las entradas son muy grande
  - Nos interesa el **"orden de crecimiento"**
  - Las constantes y términos de orden inferior no son relevantes, al ser dominados por un termino de orden superior.
- El algoritmo con mejor coste o eficiencia asintótica suele ser la mejor elección.
  - Salvo para entradas muy pequeñas



Órdenes que más aparecen

- Considerados generalmente como “tratables”

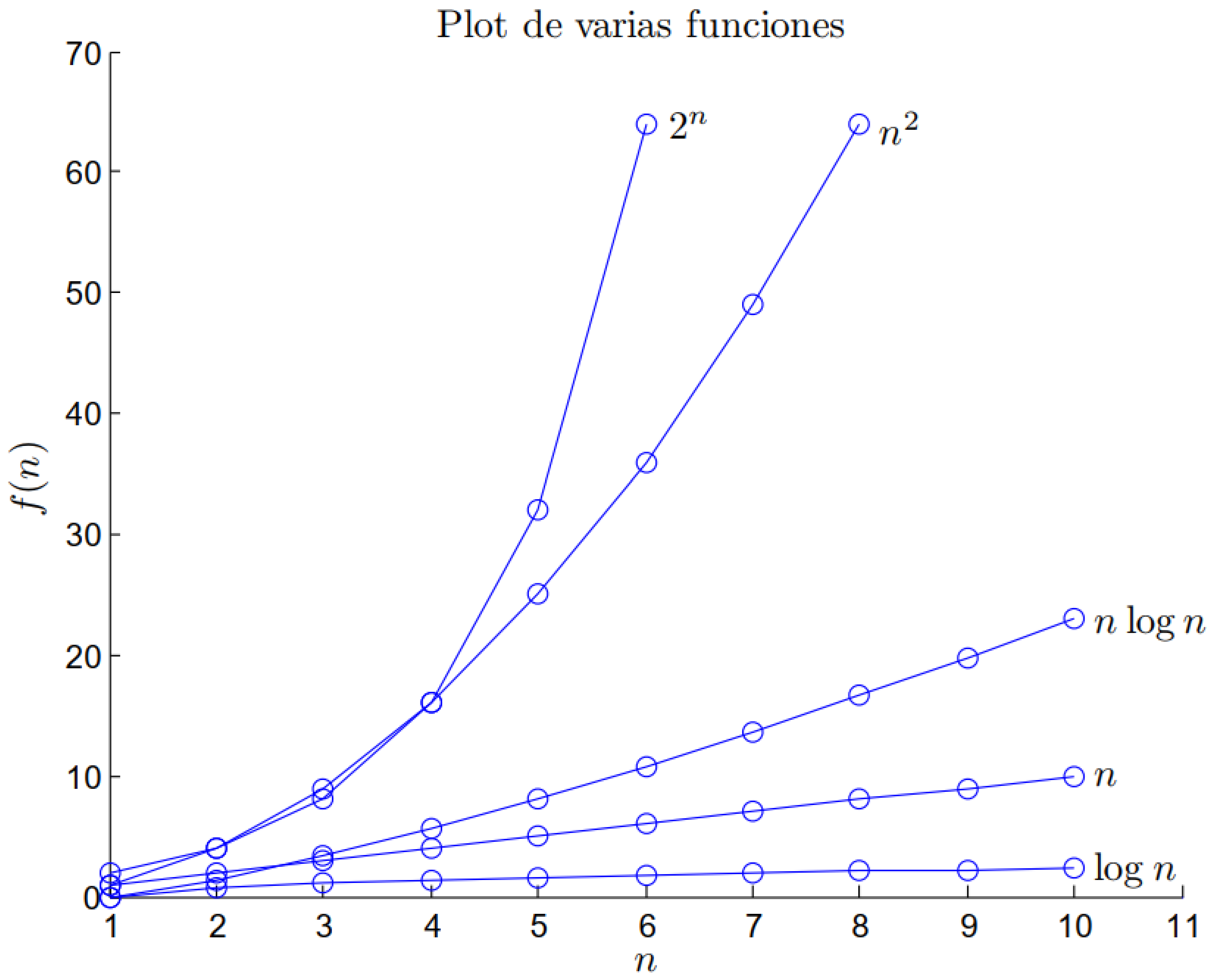
$$1 < \log n < n < n \log n < n^2$$

- Considerados generalmente como “intratables”

$$n^2 < n^3 < 2^n < n!$$

- n^2 se encuentra en el limite
- Siempre hay que tener en cuenta el tamaño de la entrada (n) para poder decir si un problema es tratable o intratable para cierto algoritmo

Órdenes que más aparecen



$\log n$	n	$n \log n$	n^2	n^3	2^n
0	1	0	1	1	2
1	2	2	4	8	4
2	4	8	16	64	16
3	8	24	64	512	256
4	16	64	256	4096	65,536
5	32	160	1024	32,768	4,294,967,296

- Un orden exponencial es extremadamente costoso, incluso frente a ordenes polinómicos.
- Un orden factorial es incluso más costoso que un orden exponencial.

¿Qué es la notación big O?

Provee un límite superior a la tasa de crecimiento de una función que representa el tiempo computacional de un algoritmo.

$$\boxed{f(n) \in \mathcal{O}(g(n)) \quad \Longleftrightarrow \quad f(n) \leq g(n)}$$

- $f(n)$ es asintóticamente menor o igual que $g(n)$
- $g(n)$ es una cota superior de $f(n)$

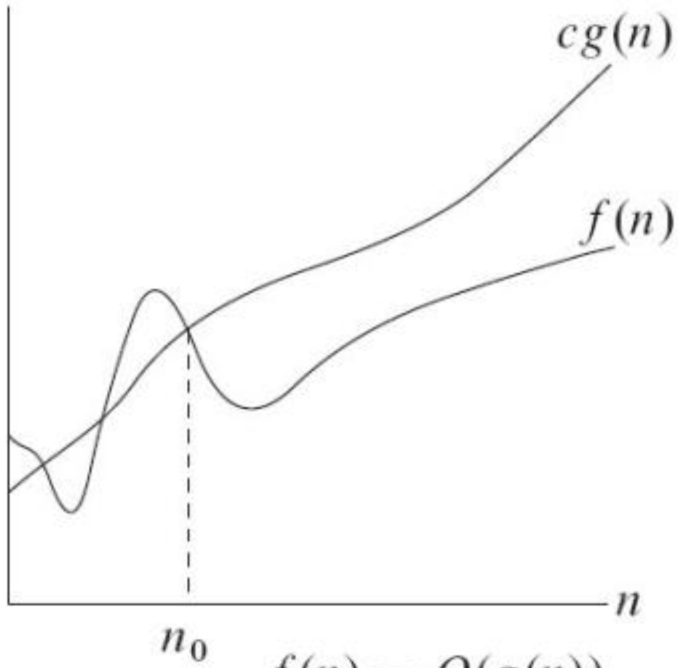
- $T = a = \boldsymbol{O}(\mathbf{1})$
- $T = an + b = \boldsymbol{O}(\boldsymbol{n})$
- $T = an^2 + bn + c = \boldsymbol{O}(\boldsymbol{n}^2)$

- $2n + 5 \in \mathcal{O}(3n^2 - 8n)$
- $2n + 5 \in \mathcal{O}(n + 10)$
- $2n + 5 \in \mathcal{O}(n!)$
- $2n + 5 \in \mathcal{O}(n)$ ← **Nos fijamos en la cota superior mas baja**

¿Qué es la notación big O?

- Definición formal

$$\mathcal{O}(g(n)) = \left\{ f(n) : \exists\ c > 0 \text{ y } n_0 > 0 \ / \ 0 \leq f(n) \leq c \cdot g(n), \forall n \geq n_0 \right\}$$



$f(n) = O(g(n))$

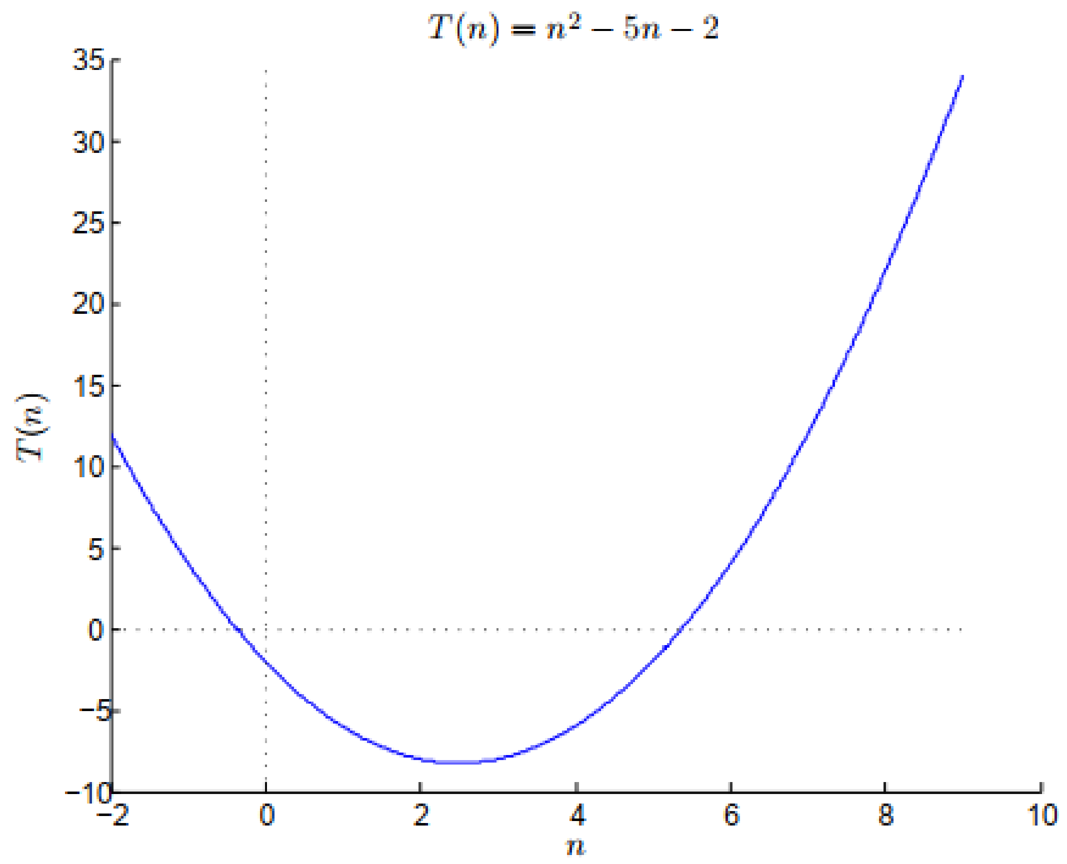
- La ide principal es que a partir de n_0, la función $c.\,g(n)$ siempre supera a $f(n)$

¿Qué es la notación big O?

- Para demostrar que una función $f\ (n)\ \in O(g(n))$ será necesario encontrar una (cualquier) pareja de constantes $c\ >\ 0$ y $n_0 >\ 0$, de tal forma que se verifiquen las condiciones de la definición.
- Ejemplo: demostrar que $\mathbf{5}\boldsymbol{n}\ +\ \mathbf{2}\ \in\ \boldsymbol{O}(\boldsymbol{n})$
  - Hay que encontrar $c\ >\ 0$ y $n_0 > 0$ tales que $5n\ +\ 2\ \leq\ cn, \forall n\ \geq n_0$
  - Para ello, elegimos una constante adecuado (por ejemplo $c = 6$)
  - Buscamos un $n > 0$ para que se cumpla $5n\ +\ 2\ \leq\ cn$
  - Encontramos que con $c\ = 6$ se cumple para todo $n\ \geq\ 2$, entonces $n_0\ = 2$.
    - Hay infinitas parejas mas, pero basta con encontrar una.

¿Qué es la notación big O?

- Ejemplo: demostrar que $\mathbf{5n + 2 \in O(n^2)}$
  - Hay que encontrar $c > 0$ y $n_0 > 0$ tales que $5n + 2 \leq cn^2, \forall n \geq n_0$
  - Elegimos $c = 1$, buscamos que valores de n cumplen $5n + 2 \leq n^2$
    - Resolvemos la desigualdad $n^2 - 5n - 2 \geq 0$
    - Las raíces de la función cuadrática son -0.37 y 5.37
    - Por lo tanto, siempre será positiva para $n_0 = 6$



- Ejemplo: demostrar que $3n^2 + 2n - 2 \in O(n)$

Cota inferior Ω

Provee un límite inferior a la tasa de crecimiento de una función que representa el tiempo computacional de un algoritmo.

$$\boxed{f(n) \in \Omega(g(n)) \iff f(n) \geq g(n)}$$

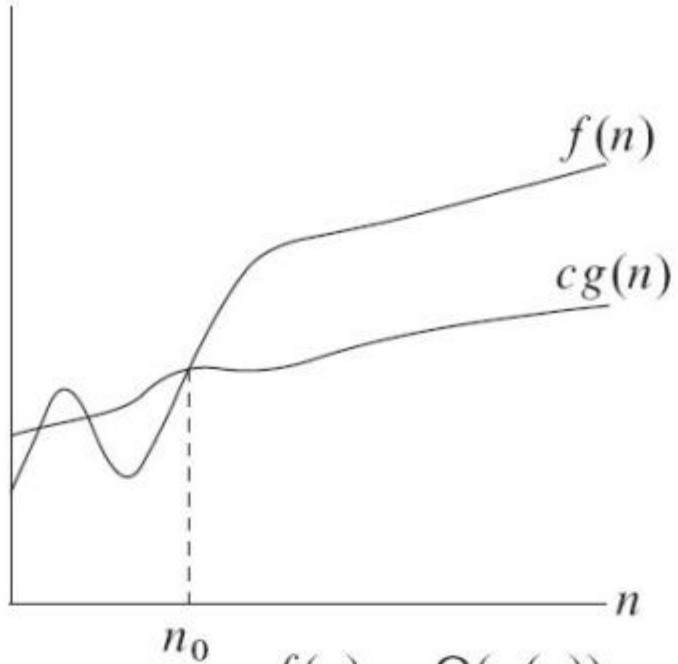
- $f(n)$ es asintóticamente mayor o igual que $g(n)$
- $g(n)$ es una cota inferior de $f(n)$

- $2n + 5 \in \Omega(3 \log n)$
- $2n + 5 \in \Omega(4n + 10)$
- $2n + 5 \in \Omega(1)$
- $2n + 5 \in \Omega(n)$ ← **Nos fijamos en la cota inferior mas alta (simplificado)**

Cota inferior Ω

- Definición formal

$$\Omega(g(n)) = \left\{ f(n) : \exists\ c > 0 \text{ y } n_0 > 0 \ / \ 0 \leq c \cdot g(n) \leq f(n), \forall n \geq n_0 \right\}$$



$f(n) = \Omega(g(n))$

- La ide principal es que a partir de n_0, la función $f(n)$ siempre supera a $c.g(n)$

Cota ajustada Θ

Provee un límite ajustado a la tasa de crecimiento de una función que representa el tiempo computacional de un algoritmo.

$$\boxed{f(n) \in \Theta(g(n)) \quad \Longleftrightarrow \quad f(n) = g(n)}$$

- $f(n)$ es asintóticamente igual que $g(n)$
- $g(n)$ es una cota ajustada de $f(n)$

Ejemplo:

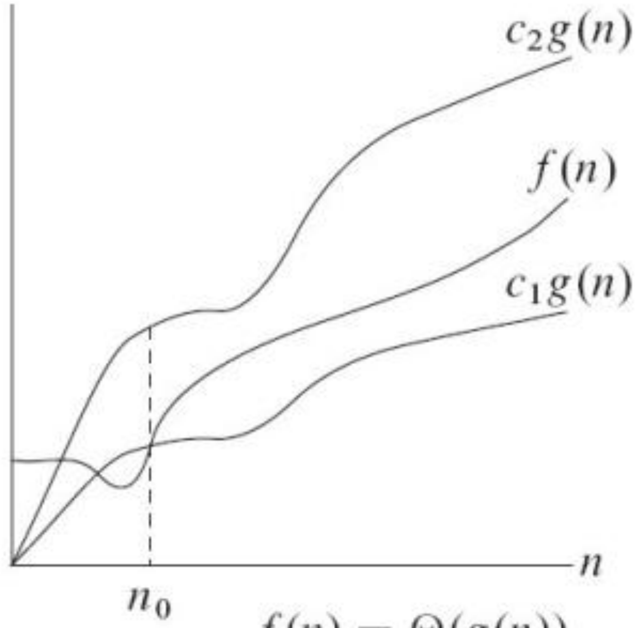
- $2n + 5 \in \Theta(8n + 10)$
- $2n + 5 \in \Theta(n)$

Cota ajustada Θ

- Definición formal

$$\Theta(g(n)) = \Big\{ f(n) : \exists\ c_1 > 0,\ c_2 > 0 \text{ y } n_0 > 0\ /$$

$$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n),\ \forall n \geq n_0 \Big\}$$



$f(n) = \Theta(g(n))$

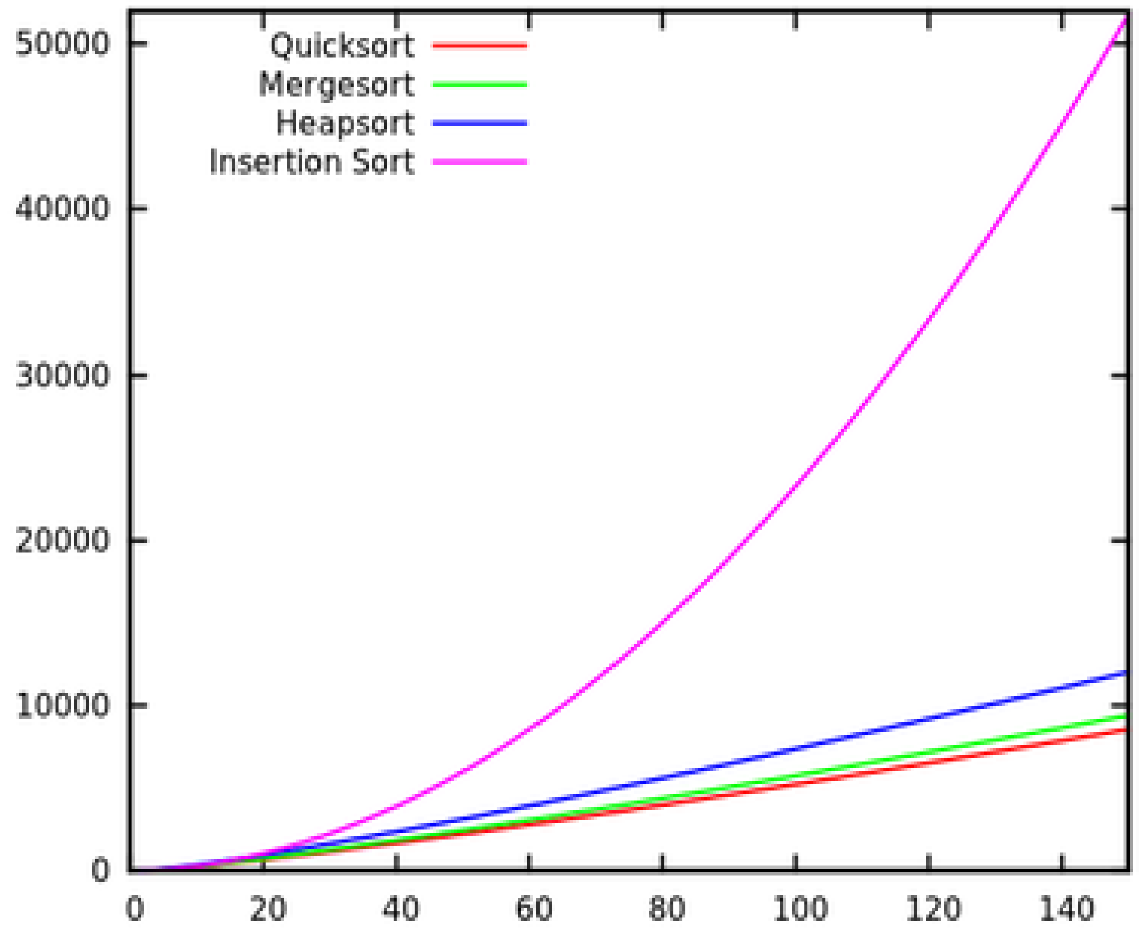
- La ide principal es que a partir de n_0, la función $f(n)$ siempre queda en medio de $c_1 . g(n)$ y $c_2 . g(n)$

Algoritmos de Ordenación

¿Qué ejemplos de ordenación se les ocurren?
Ordenar a personas en base a la edad, ordenar entradas de un blog en base a fecha, etc.

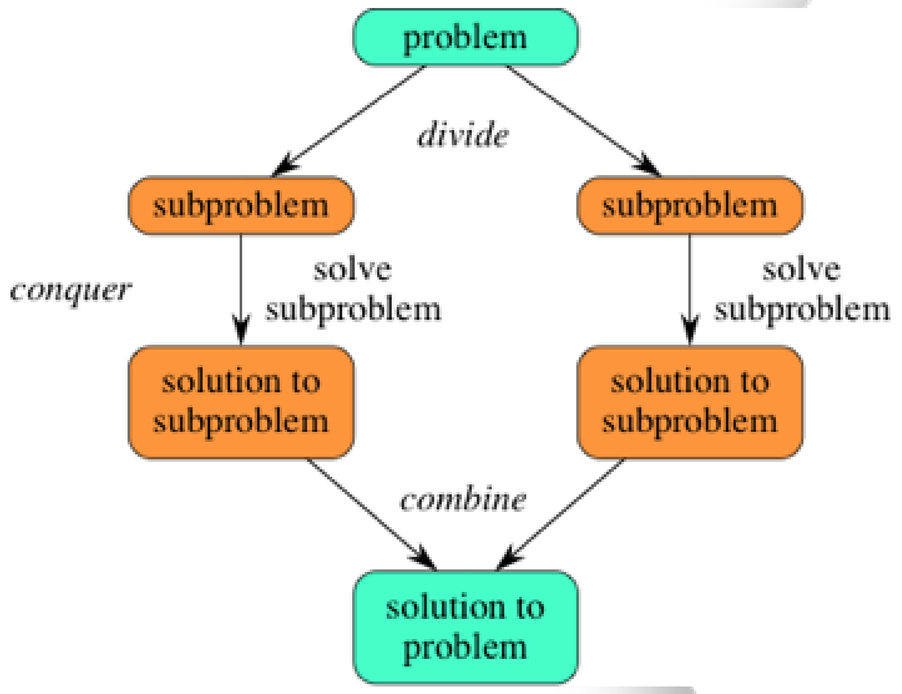
¿Qué algoritmos de ordenamiento conocen?
Bubble sort, selection sort, insert sort, quick sort, merge sort, heap sort, counting sort, etc.

¿En qué nos basamos para elegir un algoritmo de ordenación para un proyecto?
Algunos son más rápidos pero usan muchos recursos, otros lentos pero ligeros y usan pocos recursos. Esto los hace un gran caso de estudio para comparar.



Diseño de algoritmos eficientes

- **Divide y vencerás**
  - Este es un método de diseño de algoritmos que se basa en subdividir el problema en sub-problemas, resolverlos recursivamente, y luego combinar las soluciones de los sub-problemas para construir la solución del problema original.
  - Es necesario que los subproblemas tengan la misma estructura que el problema original, de modo que se pueda aplicar la recursividad.
  - Ejemplo: Mergesort, Quicksort, Multiplicación de Polinomios



Diseño de algoritmos eficientes

- **Programación dinámica**
  - Hay ocasiones en que la simple aplicación de la recursividad conduce a algoritmos muy ineficientes, pero es posible evitar esa ineficiencia con un uso adecuado de memoria.
  - **Ejemplo: Algoritmo de Fibonacci**
    - Los resultados calculados previamente se almacenan en una cache para ser usado en llamadas posteriores.

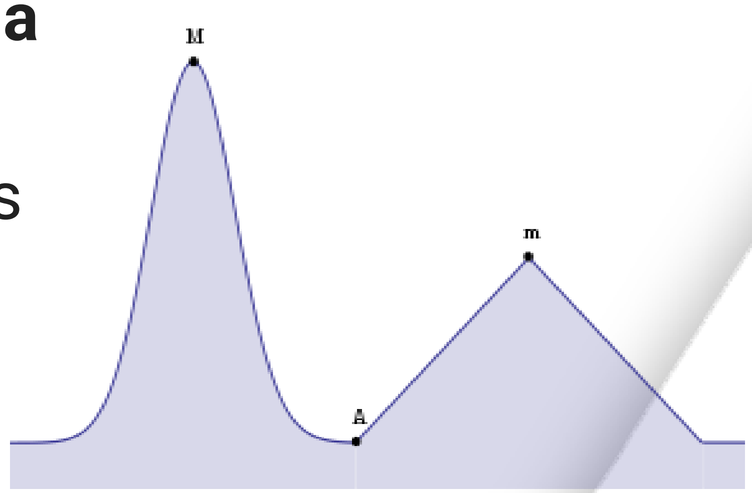
```
int fibonacci(int n){
    int cache[n+1];
    cache[0] = 0;
    cache[1] = 1;
    for (int i = 2; i <= n; i++)
        cache[i] = cache[i-1] + cache[i-2];
    return cache[n];
}
```

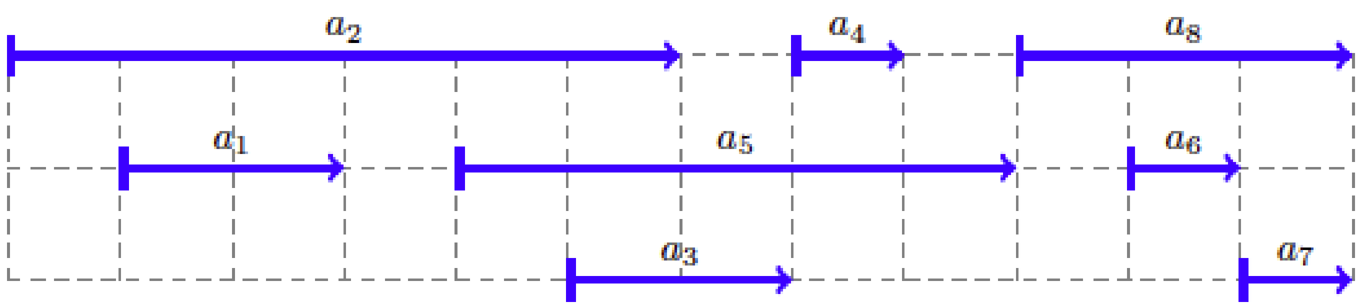
$\longrightarrow \boldsymbol{O(n)}$

- Ejemplos: Longest Common Subsequence, Levenshtein Distance, DTW

Diseño de algoritmos eficientes

- **Algoritmos voraces/avaros**
  - Un algoritmo de optimización es *avaro* si siempre toma la **decisión óptima** de **corto plazo**.
  - Por ejemplo, un algoritmo avaro que intenta llegar a la cima del cerro más alto, daría siempre un paso en la dirección que le permite subir más con ese paso.
  - En general, la estrategia avara no garantiza llegar a un óptimo global, porque es fácil quedarse atrapado en un óptimo local (en nuestro ejemplo, llegar a la cima de un cerro pequeño y no poder salir de ahí).
  - Sin embargo, hay problemas para los cuales la estrategia avara sí encuentra un óptimo global:
    - Algoritmo Earliest-Finish-First para la asignación de actividades:





- Ordenar las actividades por el tiempo final
- El algoritmo escoge la primera actividad disponible en este orden, luego elimina todas las actividades que se superponen con ella.
- Solución {a1, a3, a4, a6, a7}

UTEC
UNIVERSIDAD DE INGENIERÍA Y TECNOLOGÍA

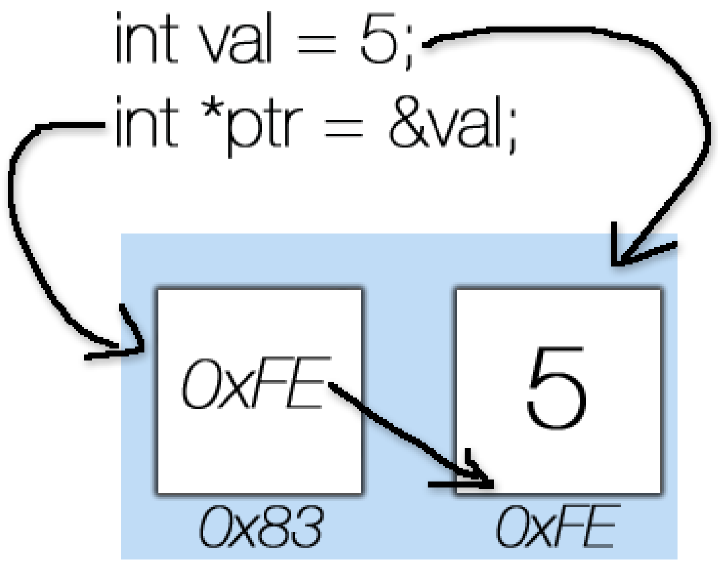
2.

Punteros

TRANSFORMATEC

Punteros

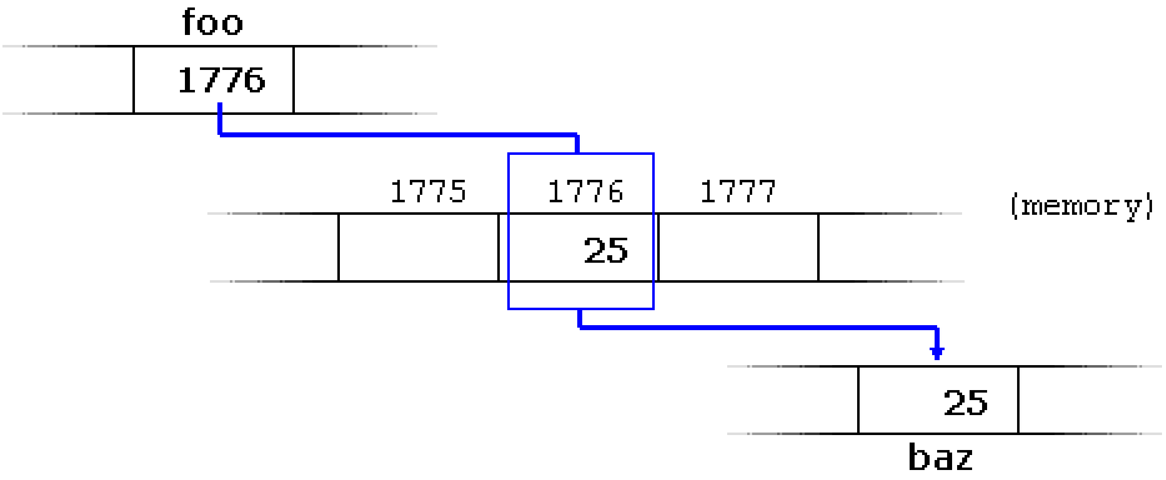
Una variable que almacena la dirección de otra variable es un puntero. Por tanto se suele decir que un puntero **“apunta a”** la variable la cual su dirección está almacenada



Operadores importantes:

Dereference (*):

a. Se utiliza para definir al puntero, y va de lado de la variable. E.g. int *a, *b, *c;
b. A su vez, permite acceder al contenido a la cual apunta. E.g. int d = *a;

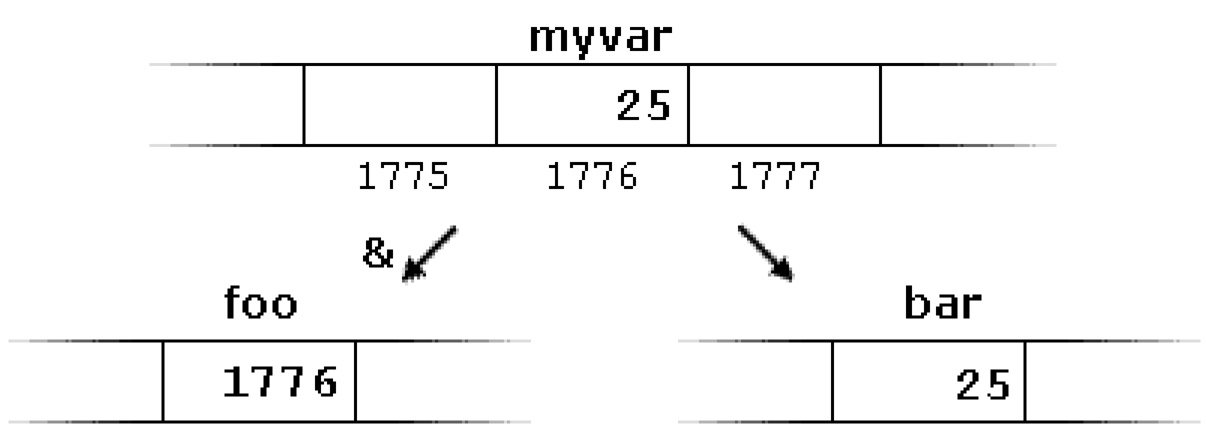


Punteros

Address-of (&):

a. Obtiene la dirección de una variable de cualquier tipo.
   E.g. int *pointer = &variable;

Por qué el tipo de dato debe ser conocido al momento de definir un puntero?
Se debe a una de las características de *Dereference,* ya que podemos obtener el valor al cual se apunta.



Recuerden que la verdadera dirección solo se puede conocer en tiempo de ejecución

Punteros

Cuando un array es declarado, se separa suficiente memoria para almacenar todos los elementos, dado:

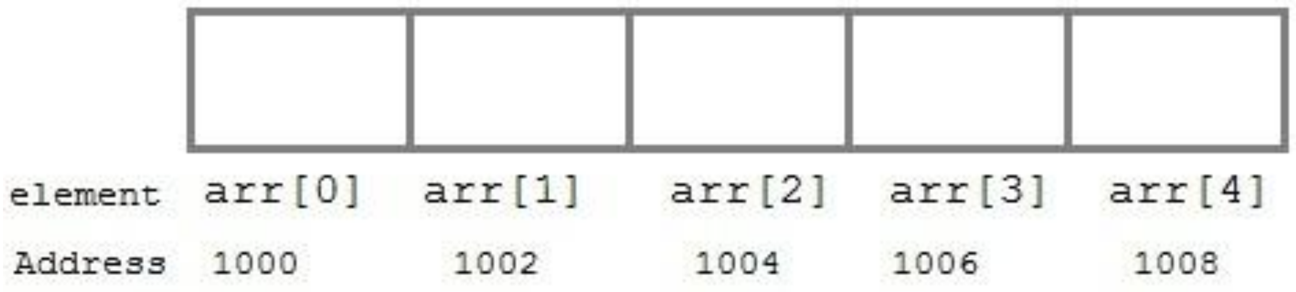
```
short arr[5] = {1, 2, 4, 8, 16};
cout << arr << endl;
```

Qué imprimirá el código anterior?

1000, ya que *arr = &arr[0]*

y esto?

```
cout << *arr + 2 << endl;
// Sería 3!
```



Punteros

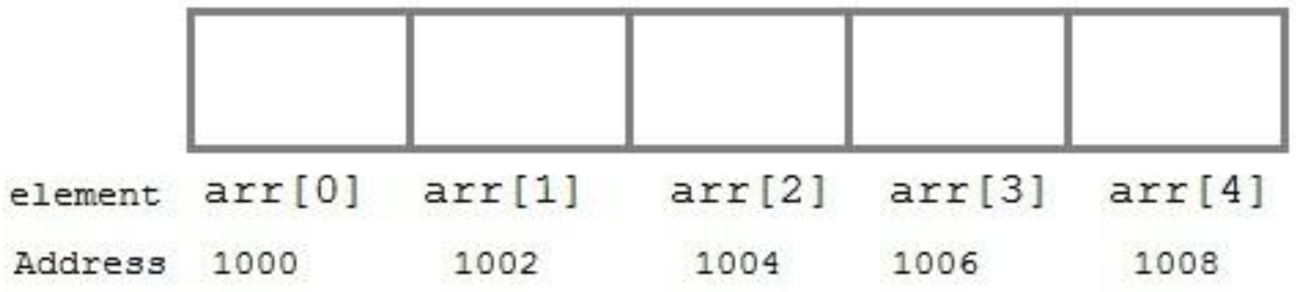
```
short arr[5] = {1, 2, 4, 8, 16};
```

Bien, entonces continuando que daría esto?

```
cout << *(arr + 2) << endl;
// Sería 4, perfecto
```

Finalmente, qué tal esto?

```
cout << *(arr + 12) << endl;
// 21908, daría basura probablemente
```



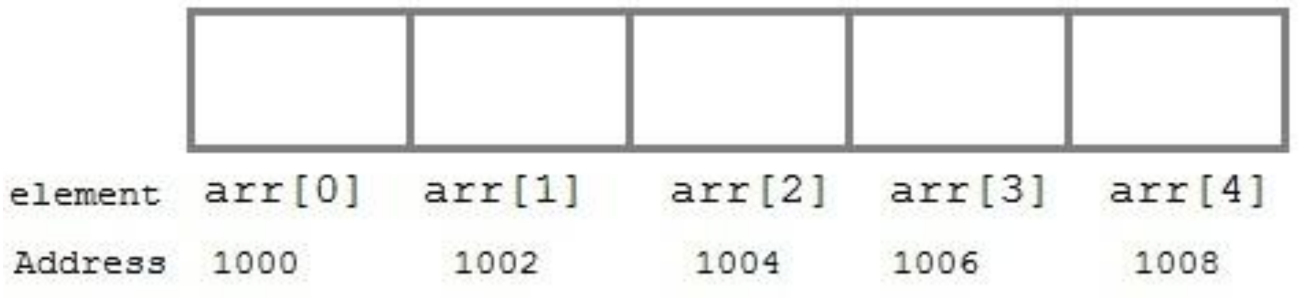
Punteros

Recuerden que un puntero y un arreglo son equivalentes en ciertas situaciones y tienen propiedades similares

int arr[5], *ptr;
ptr = arr; // Válido

Sin embargo, un puntero puede ser asignado a diferentes direccio representar el mismo bloque.

arr = ptr; // Esto nunca será válido



Punteros

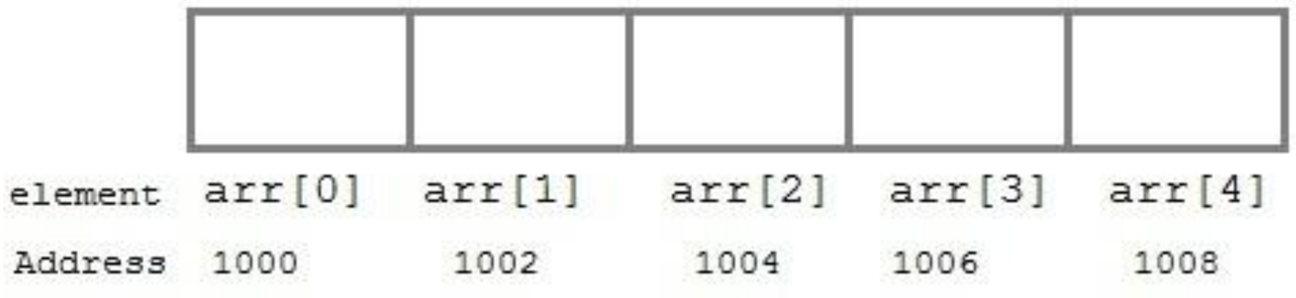
En el caso de un array de varias dimensiones, como arr[i][j], podríamos tratarlo de la siguiente manera:
***(*(arr + i) + j)**

Ahora, que imprimiría el siguiente código?

```
int number = 17, *ptr;
ptr = &number;
cout << *ptr << endl;
```

perfecto, 17. Y esto?

```
int number = 13, *ptr;
*ptr = number;
cout << *ptr << endl;
```



Error de memoria, porque el puntero no ha sido inicializado

Punteros

Cómo se pueden inicializar los punteros?

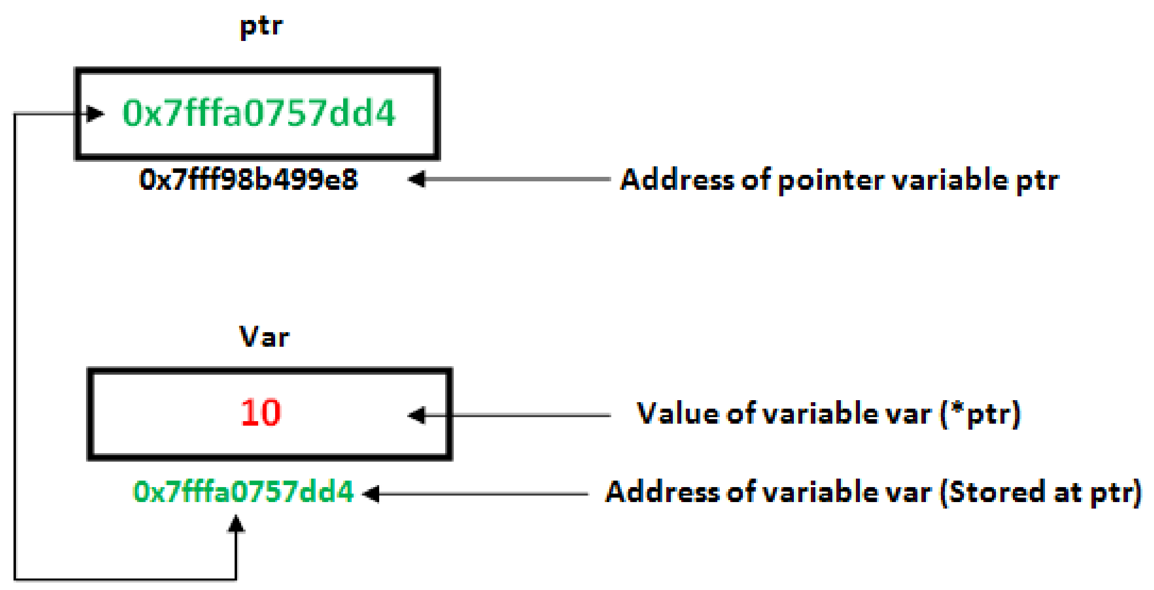
- int *ptr = new int;
  *ptr = 5;

- int variable = 5;
  int *ptr = &variable;

El siguiente código estaría correcto?

int *ptr = &5;

// No, ya que 5 no es una variable

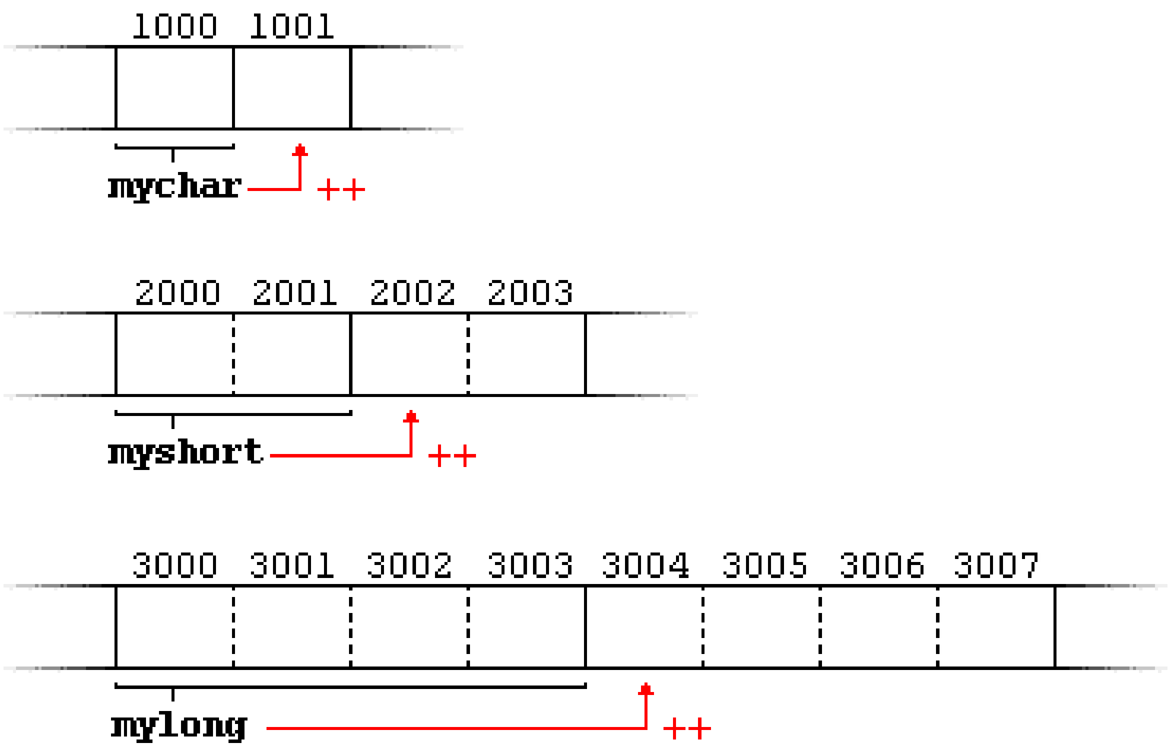


Punteros

Ya vimos un par de operaciones aritméticas sobre arreglos, me podrían decir, qué sucede aquí?

```
char *mychar;
short *myshort;
long *mylong;

++mychar;
++myshort;
++mylong;
```



Punteros

Dadas las siguientes equivalencias:

- *p++ = *(p++)
- *++p = *(++p)
- ++*p = ++(*p)

Siempre recuerden:

i++ = incrementa pero retorna original
++i = incrementa y retorna nuevo valor

Qué imprimirá?

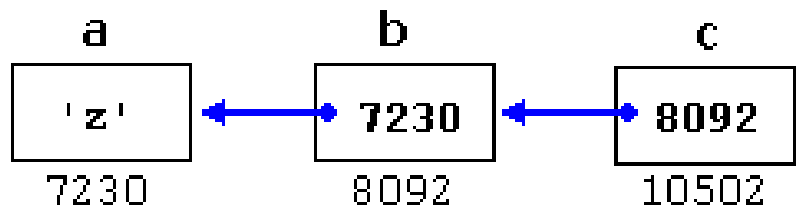
```
int arr[5] = {1, 2, 3, 4, 5};
int *ptr = arr;

cout << *ptr << endl;          1
cout << *ptr++ << endl;        1
cout << *++ptr << endl;        3
cout << ++*ptr << endl;        4
cout << (*ptr)++ << endl;      4
cout << *ptr << endl;          5
```

Punteros

Los punteros también pueden apuntar a punteros, sólo se tiene que agregar un * en la declaración: int ** ptr;

Punteros a void, son un tipo especial de punteros donde *void* representa la ausencia de tipo. Por tanto, puede ser usado para cualquier tipo con algunas restricciones en cuanto a sus operadores




```
int number = 5;
void *ptr = &number;
cout << *(int*)ptr << endl;
```

Recuerda siempre definir tus punteros a nullptr o NULL, cuando estén vacíos, de otra forma va a apuntar a basura

```
int *ptr = nullptr;
int * ptr = NULL;
```

Finalmente, un puntero también puede apuntar a funciones! Sería bueno que lo revisen

Punteros

NULL vs nullptr

NULL is a "manifest constant" (a `#define` of C) that's actually an integer that can be assigned to a pointer because of an implicit conversion.

nullptr is a keyword representing a value of self-defined type, that can convert into a pointer, but not into integers.

```
int i = NULL; // OK
int i = nullptr; // error - not a integer convertible value
int* p = NULL; //ok - int converted into pointer
int* p = nullptr; // ok
```

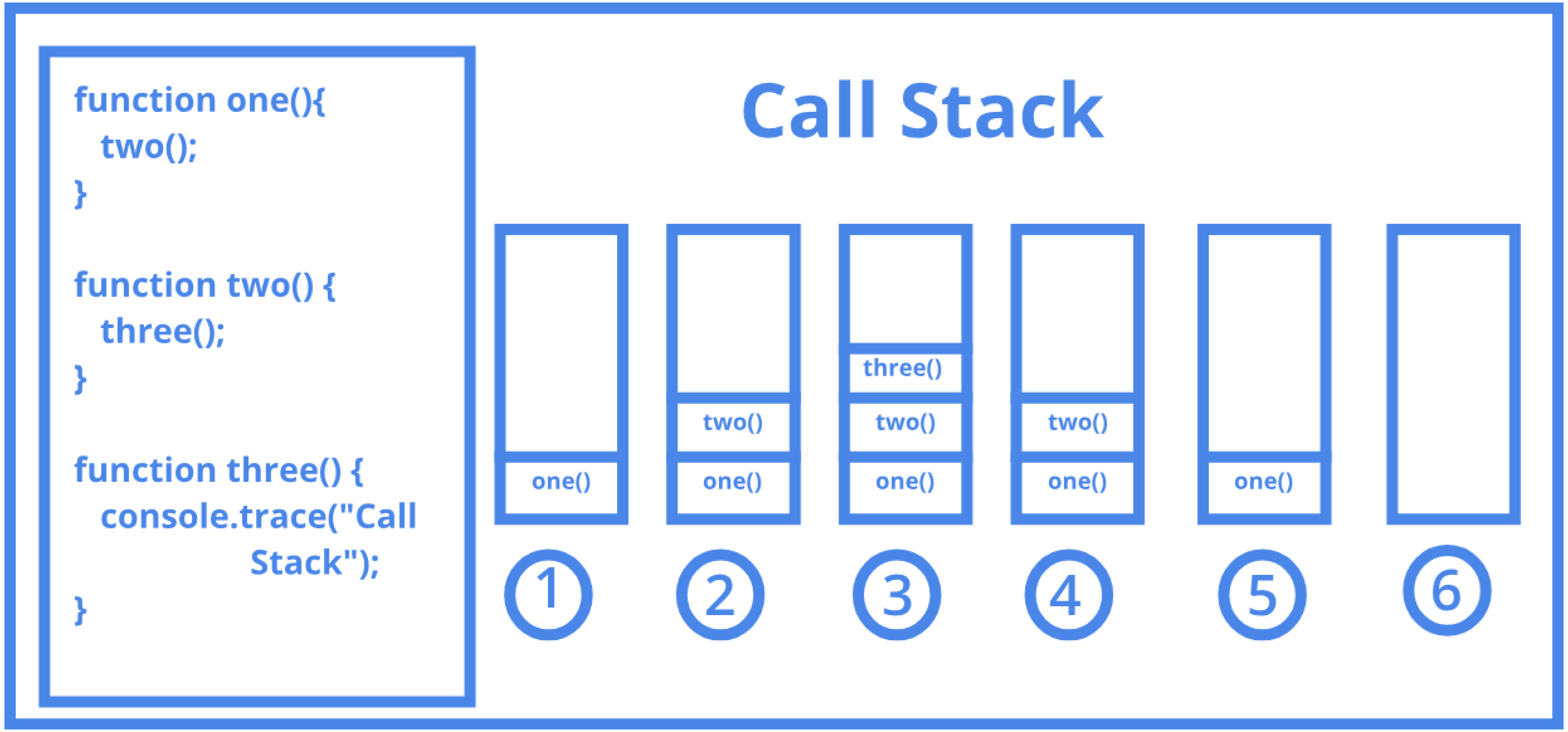
UTEC
UNIVERSIDAD DE INGENIERÍA Y TECNOLOGÍA

3.

Espacios de Memoria Stack & Heap

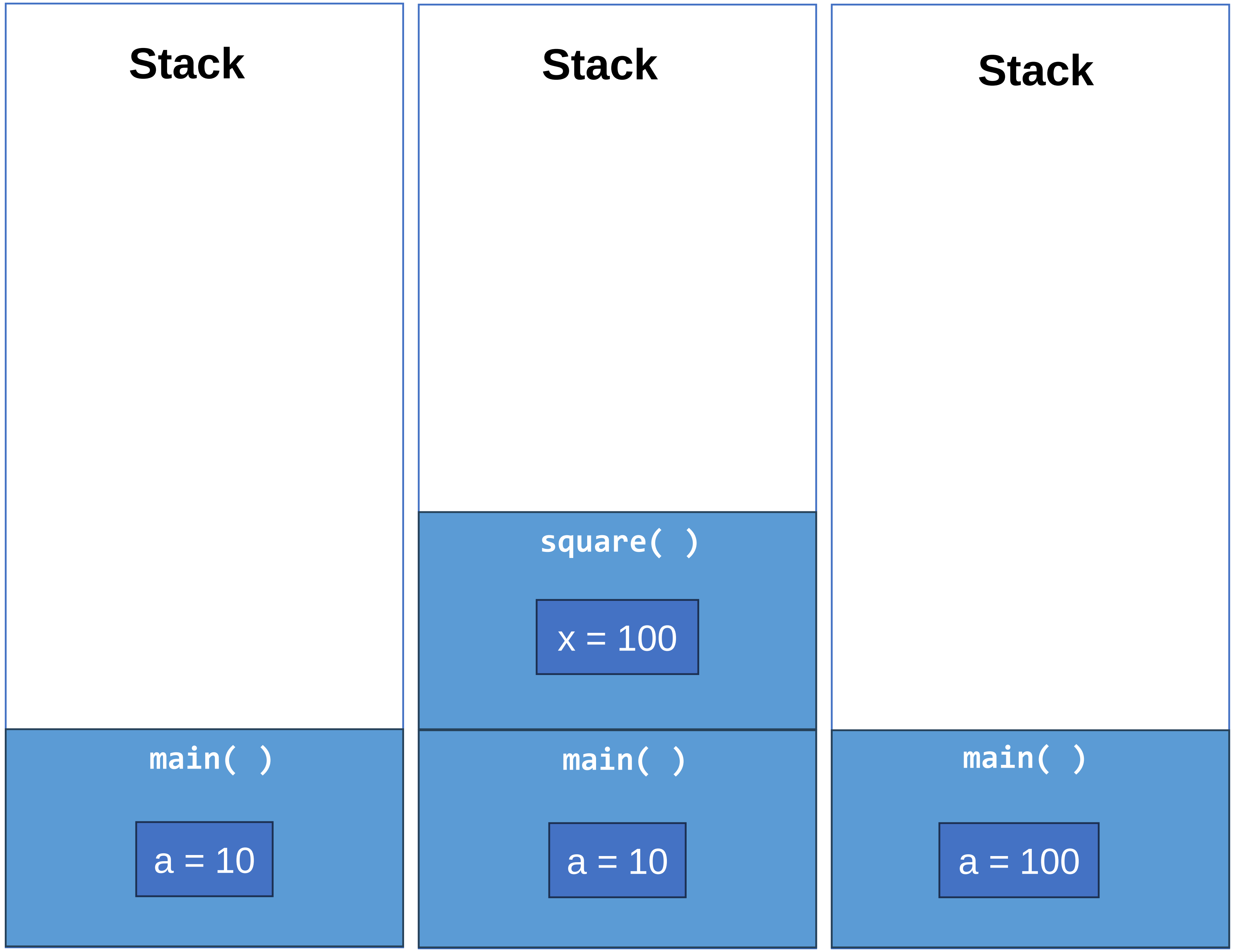
TRANSFORMATEC

Stack & Heap


function one(){
two();
}
function two() {
three();
}
function three() {
console.trace("Call
Stack");
}
Call Stack
one()
two()
one()
three()
two()
one()
two()
one()
one()
1
2
3
4
5
6

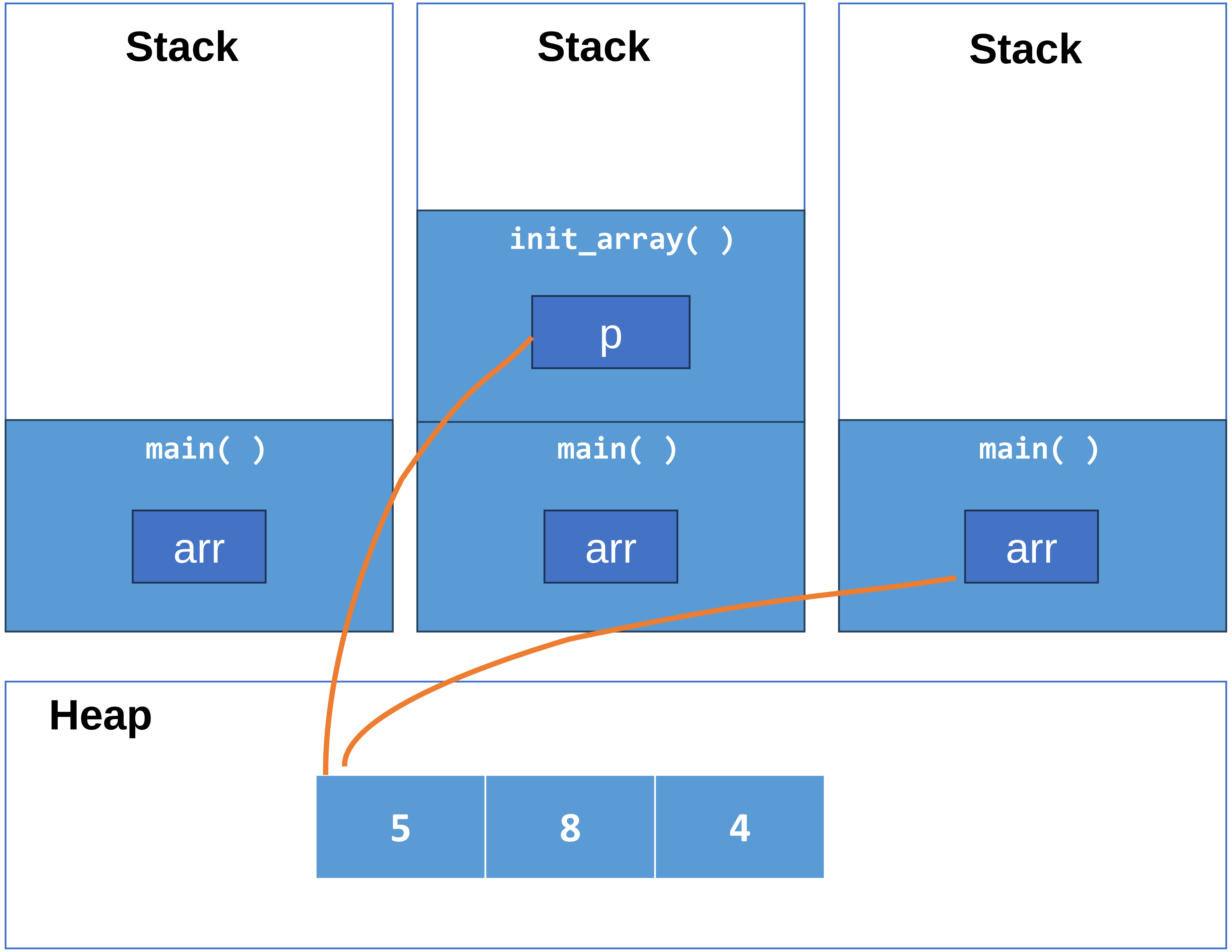
Stack & Heap

```cpp
#include <iostream>
using namespace std;
// returns the square of an
integer
int square(int x)
{
    return x * x;
}
int main()
{
    int a = 10;
    a = square(a);
    cout << a << endl;
    return 0;
}
```



Stack & Heap

```
#include <iostream>
using namespace std;
// initialize array
int* init_array()
{
    int* p =
      new int[3] {5, 8, 4};
    return p;
}
int main()
{
    int* arr = nullptr;
    arr = init_array();
    cout << arr[1] << endl;
    return 0;
}
```



Stack & Heap

```
#include <iostream>
using namespace std;
// initialize array
int* init_array()
{
    int p[3] = {5, 8, 4};
    return p;
}
int main()
{
    int* arr = nullptr;
    arr = init_array();
    cout << arr[1] << endl;
    return 0;
}
```

¿Qué ocurre aquí?

UTEC
UNIVERSIDAD DE INGENIERÍA Y TECNOLOGÍA

> Reinventa el mundo <

GRACIAS

Heider Sanchez

TRANSFORMATEC