

# CS2013: Programación III

Teoría: Complejidad Algorítmica I

José Chávez



# Agenda

- Invariante de bucle
- Prueba de correctitud
- Notación Big-O





```
void insertion_sort(std::vector<int>& v){
    for(int j = 1; j < v.size(); j++){
        int key = v[j];
        int i = j-1;
        while(i >= 0 && v[i] > key){
            v[i+1] = v[i];
            i--;
        }
        v[i+1] = key;
    }
}
```



```
void insertion_sort(std::vector<int>& v) {
    for(int j = 1; j < v.size(); j++) {
        int key = v[j];
        int i = j-1;
        while(i >= 0 && v[i] > key) {
            v[i+1] = v[i];
            i--;
        }
        v[i+1] = key;
    }
    El elemento actual a ser insertado
}
```



```
void insertion_sort(std::vector<int>& v){
    for(int j = 1; j < v.size(); j++){
        int key = v[j];
        int i = j-1;
        while(i >= 0 && v[i] > key){
            v[i+1] = v[i];
            i--;
        }
        v[i+1] = key;
}
```



```
void insertion_sort(std::vector<int>& v) {
    for(int j = 1; j < v.size(); j++) {
        int key = v[j];
        int i = j-1;
        while(i >= 0 && v[i] > key) {
            v[i+1] = v[i];
            i--;
        }
        v[i+1] = key;
    }
}
```



A[0,,j-1]	key	Secuencia Desordenada
		5,3,2,1,4,6



A[0,,j-1]	key	Secuencia Desordenada
5	3	2,1,4,6

$$key = v[1]$$



A[0,,j-1]	key	Secuencia Desordenada
3,5		2,1,4,6



A[0,,j-1]	key	Secuencia Desordenada
3,5	2	1,4,6



A[0,,j-1]	key	Secuencia Desordenada
2,3,5		1,4,6



A[0,,j-1]	key	Secuencia Desordenada
2,3,5	1	4,6



A[0,,j-1]	key	Secuencia Desordenada
1,2,3,5		4,6



A[0,,j-1]	key	Secuencia Desordenada
1,2,3,5	4	6



A[0,,j-1]	key	Secuencia Desordenada
1,2,3,4,5		6

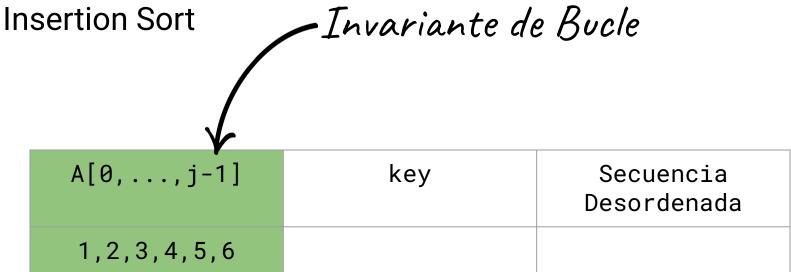


A[0,,j-1]	key	Secuencia Desordenada
1,2,3,4,5	6	

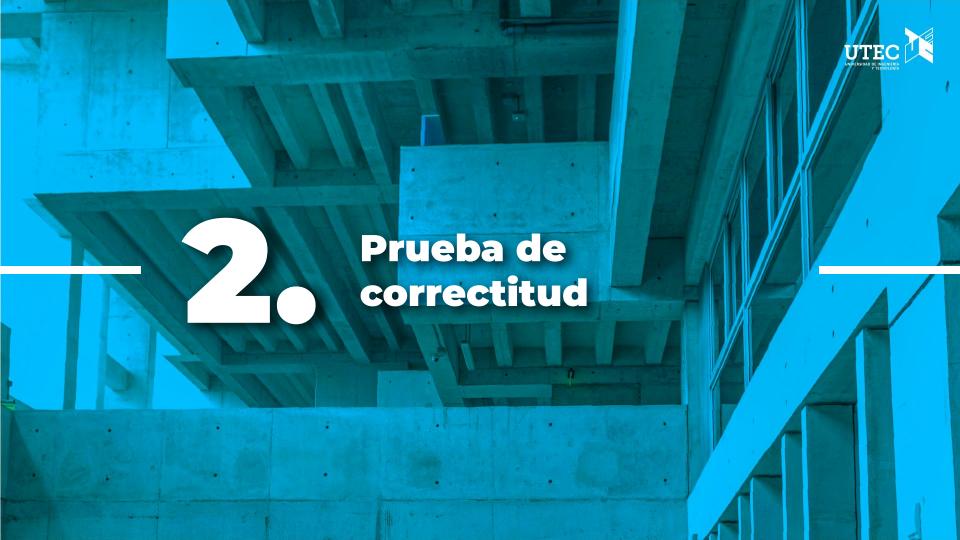


A[0,,j-1]	key	Secuencia Desordenada
1,2,3,4,5,6		





El invariante de bucle es una condición que se cumple en cada iteración del bucle





#### Prueba de correctitud

Para demostrar la correctitud de un algoritmo podemos utilizar el invariante de bucle. Este debe cumplir tres propiedades:

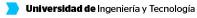
- Inicialización
- 2. Mantenimiento
- 3. Terminación



#### Inicialización

```
void insertion_sort(std::vector<int>& v){
    for(int j = 1; j < v.size(); j++){
        int key = v[j];
        int i = j-1;
        while(i >= 0 && v[i] > key){
            v[i+1] = v[i];
            i--;
        }
        v[i+1] = key;
    }
}
```

El invariante de bucle se debe cumple antes de la primera iteración





#### Inicialización

```
void insertion_sort(std::vector<int>& v){
    for(int j = 1; j < v.size(); j++){
        int key = v[j];
        int i = j-1;
        while(i >= 0 && v[i] > key){
            v[i+1] = v[i];
            i--;
        }
        v[i+1] = key;
    }
}
```

Cuando j=1, el sub-conjunto A[0,...,j-1] está ordenado porque lo conforma solo un elemento

El invariante de bucle se debe cumple antes de la primera iteración





#### Mantenimiento

```
void insertion_sort(std::vector<int>& v){
    for(int j = 1; j < v.size(); j++){</pre>
        int key = v[i];
        int i = j-1;
        while(i >= 0 && v[i] > key){
            v[i+1] = v[i]:
```

Si A[0,...,j-1] ya está ordenado, estas líneas insertan el elemento A [ j ] en la posición correcta

Si el invariante de bucle se cumple antes de un iteración, entonces se debe mantener al final de la iteración



#### **Terminación**

```
void insertion_sort(std::vector<int>& v){
    for(int j = 1; j < v.size(); j++){
        int key = v[j];
        int i = j-1;
        while(i >= 0 && v[i] > key){
            v[i+1] = v[i];
            i--;
        }
        v[i+1] = key;
    }
}
```

```
Al terminar el bucle, cando j=v.size(), la secuencia [0,...,v.size()-1] resulta ordenada
```

Al terminar el bucle, el invariante de bucle ayuda a demostrar la correctitud el algoritmo



#### **Terminación**

```
void insertion_sort(std::vector<int>& v) {
    for(int j = 1; j < v.size(); j++) {
        int key = v[j];
        int i = j-1;
        while(i >= 0 && v[i] > key) {
            v[i+1] = v[i];
            i--;
        }
        v[i+1] = key;
    }
}
```

Al terminar el bucle, el invariante de bucle ayuda a demostrar la correctitud el algoritmo



#### Se demuestra la correctitud

```
void insertion_sort(std::vector<int>& v){
    for(int j = 1; j < v.size(); j++){
        int key = v[j];
        int i = j-1;
        while(i >= 0 && v[i] > key){
            v[i+1] = v[i];
            i--;
        }
        v[i+1] = key;
    }
}
```

Al cumplirse las tres propiedades se demuestra la correctitud del algoritmo







La notación Big-O es utilizada para determinar la eficiencia de un algoritmo.



La notación Big-O es utilizada para determinar la eficiencia de un algoritmo.

Podemos decir que:

Mientras más rápido es un algoritmo, menos trabajo hace.



La notación Big-O es utilizada para determinar la eficiencia de un algoritmo.

#### Podemos decir que:

- Mientras más rápido es un algoritmo, menos trabajo hace.
- Mientras más lento es un algoritmo, más trabajo hace.



La notación Big-O es utilizada para determinar la eficiencia de un algoritmo.

#### Podemos decir que:

- Mientras más rápido es un algoritmo, menos trabajo hace.
- Mientras más lento es un algoritmo, más trabajo hace.

#### Ejemplo:

Si para resolver un mismo problema, el algoritmo 1 lo realiza en un tiempo O(n) y el algoritmo 2 en  $O(n^2)$ . Entonces el primer algoritmo es más rápido que el segundo.



La notación Big-O representa una cota superior para una función:

Si escribimos que f(n) = O(g(n)), entonces:

$$0 \le f(n) \le cg(n)$$

Para  $n \ge n_0$ , donde  $n_0$  y c son números positivos.





# Tiempo Constante

Sin importar el tipo de entrada o tamaño, al algoritmo le tomará el mismo tiempo realizar el mismo trabajo.



# Tiempo Constante

Sin importar el tipo de entrada o tamaño, al algoritmo le tomará el mismo tiempo realizar el mismo trabajo.

#### Ejemplo:

```
POSNEG(n):
1. if n > 0
2. imprimir "n es positivo"
3. else
4. imprimir "n no es positivo"
```



# Algoritmos de tiempo constante

Si un algoritmo realiza **un número fijo de pasos** para resolver un problema, no importa si es 1, 10, 1000 o 1M, este seguirá siendo de tiempo constante.

Notación: T(n) = O(1)

### Ejemplo:

Si el algoritmo requiere 100 pasos para resolver un problema, entonces T(n)=100m. En otras palabras, T(n)=O(1).





# Tiempo Lineal

La gran mayoría de los algoritmos no son de tiempo constante. Habitualmente, el trabajo que realiza un algoritmo depende del tamaño de la entrada.

**Trabajo** de procesar 1000 elementos > **Trabajo** de procesar 10 elementos

En el caso que un algoritmo de tiempo lineal se ejecute primero con una entrada de 50 elementos y luego con otra 100 elementos. El algoritmo realizará el doble de trabajo en segundo caso.



## Tiempo Lineal

Un algoritmo de tiempo lineal tienen relación directa con el tamaño de la entrada

## Ejemplo:

```
SUMADEUNOS (n):
```

- 1. s = 0
- 2. **for** i=1 to n
- 3. s = s + 1
- 4. return s

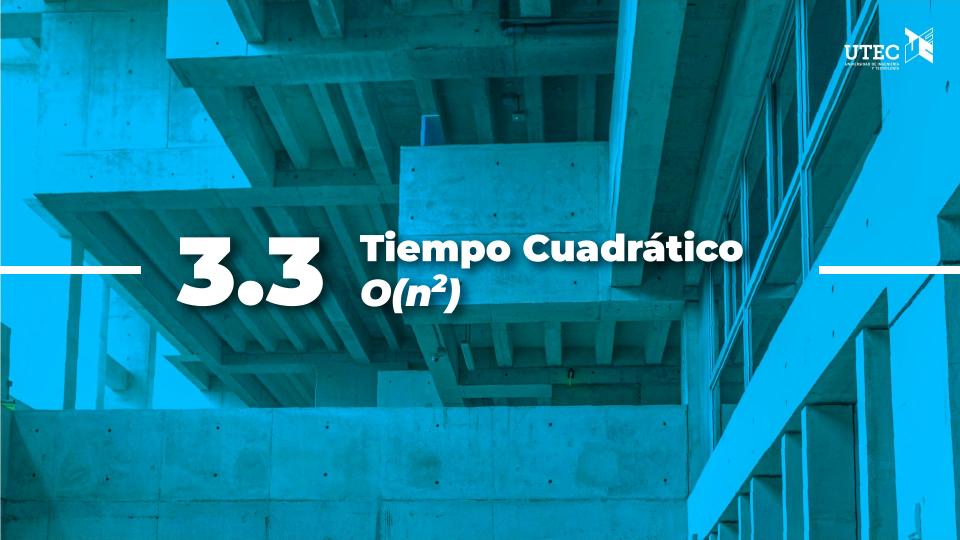


# Algoritmos de tiempo lineal

Notación: T(n) = O(n)

## Ejemplo:

- Si el algoritmo requiere 10n pasos para resolver un problema, entonces T(n)=10n. En otras palabras, T(n)=O(n).
- Si T(n)=10n + 5, entonces T(n) = O(n).





# Tiempo Cuadrático

En un algoritmo de tiempo cuadrático el trabajo realizado por el algoritmo también depende del tamaño de la entrada. Estos lucen de la siguiente manera:

```
1. for i=0 to n-1
```

2. **for** 
$$j=0$$
 **to**  $n-1$ 

3. sentencia



# Tiempo Lineal vs Tiempo Cuadrático

Si para resolver un problema, un algoritmo necesita  $n^2$  pasos para revolverlo y otro algoritmo requiere n pasos, ¿Cuál escogería?

Supongamos que se tiene 100 elementos, entonces:

- El algoritmo de tiempo lineal realizará 100 pasos para resolver el problema.
- El algoritmo de tiempo cuadrático necesitará 10000 pasos, lo cual es más costoso.





# Tiempo Logarítmico

¿Cuál sería la forma más eficiente de buscar un elemento de una secuencia de números ordenada?



#### Opción 1:

```
BUSQUEDALINEAL(L, e)

1. n = size(L)

2. for i=0 to n-1

3. if L[i] == e

4. return i

5. return -1
```



#### Opción 1:

```
BUSQUEDALINEAL (L, e)

1. n = size(L)

2. for i=0 to n-1

3. if L[i] == e

4. return i

5. return -1
```

¿Cuál es la complejidad del algoritmo?



#### Opción 1:

```
BUSQUEDALINEAL (L, e)

1. n = size(L)

2. for i=0 to n-1

3. if L[i] == e

4. return i

5. return -1
```

¿Cuál es la complejidad del algoritmo?

Rpta. O(n)





#### Opción 1:

```
BUSQUEDALINEAL (L, e)

1. n = size(L)

2. for i=0 to n-1

3. if L[i] == e

4. return i

5. return -1
```

¿Podríamos mejorar este algoritmo?

¿Cuál es la complejidad del algoritmo?

Rpta. O(n)





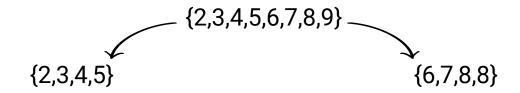
### Opción 2:

Supongamos que L =  $\{2,3,4,5,6,7,8,9\}$ , entonces



#### Opción 2:

Supongamos que  $L = \{2,3,4,5,6,7,8,9\}$ , entonces

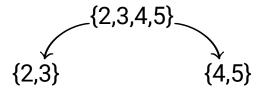


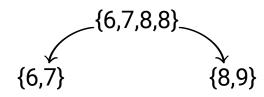


### Opción 2:

Supongamos que  $L = \{2,3,4,5,6,7,8,9\}$ , entonces

{2,3,4,5,6,7,8,9}







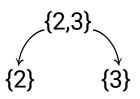
## Opción 2:

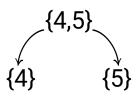
Supongamos que  $L = \{2,3,4,5,6,7,8,9\}$ , entonces

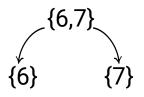
{2,3,4,5,6,7,8,9}

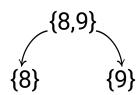
{2,3,4,5}

{6,7,8,8}











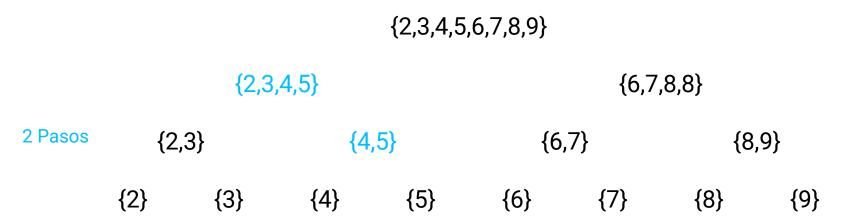
### Opción 2:



#### Opción 2:



### Opción 2:





### Opción 2:



### Opción 2:

Ahora busquemos el número 4 en la secuencia L:

{2,3,4,5,6,7,8,9}

{6,7,8,8}

{2,3}

{4,5}

{6,7}

{8,9}

3 Pasos

{3}

**{4**}

{5}

**{6**}

{7}

{8}

{9}



#### Opción 2:

```
BUSQUEDABINARIA(L, e, inicio, final)
1. if inicio <= final
       medio = (inicio + final)// 2
3.
       if L[medio]==e
4.
           return medio
5. else if (L[medio] > e)
6.
            return BUSQUEDABINARIA(L, e, inicio, medio-1)
       else if (L[medio] < e)</pre>
7.
            return BUSQUEDABINARIA(L, e, medio+1, final)
8.
    return -1
```



### Opción 2:

```
BUSQUEDABINARIA(L, e, inicio, final)
                                            T(n)=O(\log n)
1. if inicio <= final
       medio = (inicio + final)// 2
3.
        if L[medio]==e
4.
            return medio
5. else if (L[medio] > e)
6.
            return BUSQUEDABINARIA(L, e, inicio, medio-1)
7.
       else if (L[medio] < e)</pre>
            return BUSQUEDABINARIA(L, e, medio+1, final)
8.
    return -1
```





#### Resumen

En esta sesión se trataron los tópicos siguientes:

- El invariante de bucle
- Prueba de correctitud de algoritmos
- Notación Big-O
- Algoritmos de tiempo constante, lineal, cuadrático y logarítmico.



