Escuela Profesional de
Ciencia de la Computación

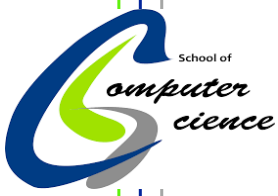
Algoritmos y Estructuras de Datos

2020-B

Algoritmos y Estructuras de Datos

M.Sc. Franci Suni Lopez

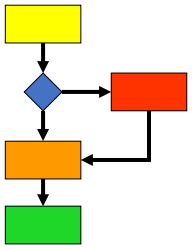
Universidad Nacional de San Agustín de Arequipa



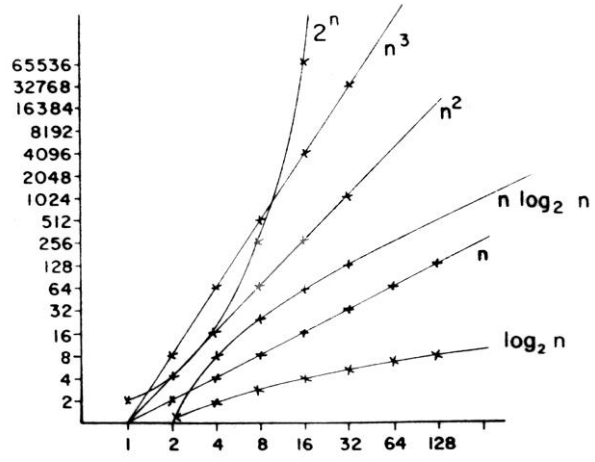
COMPLEJIDAD DE ALGORITMOS

Algoritmo

- Un algoritmo es un procedimiento computacional bien definido, que toma un conjunto de valores de entrada y produce valores de salida.
- También podemos ver a un algoritmo como una herramienta para resolver un problema computacional específico.



Complejidad



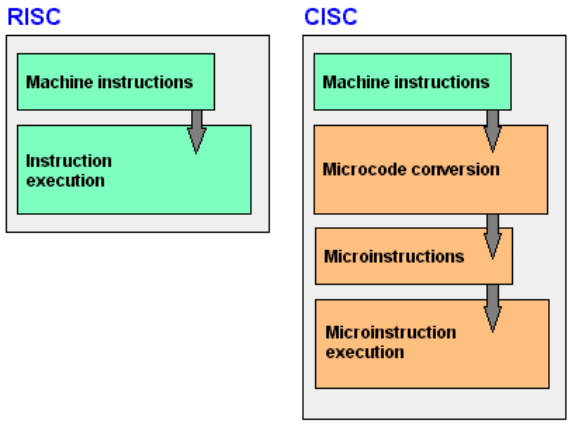
- La complejidad de un algoritmo es una medida de la cantidad de recursos que consume.
- Recurso:
  - Tiempo
  - Espacio
    - Memoria
    - Disco

Complejidad de Algoritmos

- La *Eficiencia* nos la da el *Análisis de Algoritmos:*
  - *Dimensión Temporal: Medida del* ***tiempo empleado.***
  - *Dimensión Espacial: medida de los* ***recursos invertidos.***
- Encontrar Algoritmos eficientes puede definir si Existe o no una Solución al Problema.
- Al Análisis de Algoritmos se centra en el estudio de los Bucles, del cual en última instancia, dependerán las instrucciones a ser ejecutadas.
  - No se puede realizar un análisis del número de Instrucciones pues son dependientes de las tecnologías (RISK, CISC).

$$Eficiencia = F(n)$$

siendo **n** la cantidad de elementos a ser procesados



RISC (reduced instruction set computer)

CISC (complex instruction set computer)

ARQUITECTURAS ASPECTOS	RISC	CISC
Significado	Computadoras con un conjunto de instrucciones complejo	Computadoras con un conjunto de instrucciones reducido.
Aplicación	Utilizada para entornos de red	Aplicada en ordenadores domésticos
Características	Instrucciones de tamaño fijo. Sólo las instrucciones de carga y almacenamiento acceden a la memoria de datos.	Instrucciones muy amplias.
Objetivos	Posibilitar la segmentación y el paralelismo en la ejecución de instrucciones y reducir los accesos a memoria.	Permitir operaciones complejas entre operandos situados en la memoria o en los registros internos
Ventajas	La CPU trabaja más rápido al utilizar menos ciclos de reloj. Reduciendo la ejecución de las operaciones. Cada instrucción puede ser ejecutada en un solo ciclo del CPU	Reduce la dificultad de crear compiladores. Permite reducir el costo total del sistema. Mejora la compactación de código. Facilita la depuración de errores.
Microprocesadores Basados en:	• **Intel** 8086, 8088, 80286, 80386, 80486. • **Motorola** 68000, 68010, 68020, 68030, 6840	MIPS Technologies IBM POWER PowerPC de Motorola e IBM SPARC y UltraSPARC

Ejemplo

```
15      for (size_t i = 0; i < n; i++)
16          cout<<i<<" ";
17      for (size_t i = 0; i < n; i++)
18          cout<<i<<" ";
```

```
21      for (size_t i = 0; i < n; i++)
22      {
23          cout<<"****ITERATION "<<i+1<<"****"<<endl;
24          for (size_t k = 0; k < n; k++)
25              cout<<"****BAT "<<k+1<<"****"<<endl;
26      }
```

Utilidad del Análisis

Cuando desarrollamos un algoritmo nos interesados en:

- Tener un idea qué tan bien puede esperarse que trabaje
- Cómo se compararía con el desempeño de otros algoritmos en el mismo problema.

- El análisis de un algoritmo nos puede ayudar a comprenderlo mejor, y nos puede sugerir mejoras.

- Entre más complicado sea un algoritmo, más complicado es su análisis.

Ejemplo Práctico 1

Calcular la complejidad de un algoritmo para encontrar el mayor valor en un arreglo de tamaño N.

```
class buscador {

   int busca_max(int L[]) {
      int Max = L[0];

      for (int i = 1; i < L.length; i++) {
         if (L[i] > Max) {
            Max = L[i];
          }
       }
      return Max;
   }
}
void main(String[] args) {
   int L[] = { 3, 6, 1, 8, 4, 2, 8, 4, 9, 23, 5, 3 };
   int Max;
   Max = busca_max(L);
   System.out.println(Max);
 }
```

Ejemplo Práctico 1

- En el mejor de los casos
- En el caso promedio
- En el peor de los casos

Complejidad

- ¿Qué medimos?:
  - Tiempo de ejecución
- ¿De qué depende?:
  - Tamaño de la entrada
  - Otros factores (HW y SW)
    - Velocidad de la máquina
    - Calidad del compilador
    - Calidad del programa
- ¿ Cómo medirlo?:
  - Experimentalmente
  - Estimándolo matemáticamente

Medición de la Eficiencia

- Una buena estimación de la Eficiencia debe ser independiente de aspectos de implementación (la plataforma y el Lenguaje con que se codificará) pero debe reflejar sus diferencias.
- Usamos el Orden de Magnitud de la Eficiencia del Tiempo y de los Recursos.
- Se lo complementa con la Tasa de crecimiento del programa para evaluar su comportamiento a Futuro.

1. Medición Experimental

- Midiendo el tiempo de ejecución en función del tamaño de la entrada
  - No podemos probar con todas las entradas
  - Es necesario implementar el algoritmo
  - Depende del software y el hardware
- Busquemos otra medida
  - Más abstracta
  - Más fácil de obtener

Medición de la Eficiencia

- ¿Cómo comparar la Eficiencia Real de 2 Algoritmos?
  - Programándolos, implementándolos y midiendo para el mismo Lote de Prueba: el tiempo de ejecución y los recursos utilizados.
  - Suele ser caro, y una vez realizado no suele ser sencillo volver atrás y corregir

2. Estimación Matemática

- Asociamos a cada algoritmo una función f(n)
- Emplearemos el caso peor para caracterizar el tiempo de ejecución
- *Es más fácil de calcular*
- Interesa la velocidad de crecimiento del tiempo de ejecución en función del tamaño de la entrada
- Comportamiento asintótico
- Asociamos a cada operación primitiva un tiempo de ejecución constante:
  - Asignación
  - Operación aritmética
  - Seguir una referencia
  - Etc.
  - Llamada a método
  - Indexación en array
  - Volver de un método

Ejemplo Práctico 2

Calcular la complejidad de un algoritmo para calcular el factorial de un número n.

```
int iterative(int n) {
   int fact = 1;
   for (int i = 1; i <= n; i++)
      fact *= i;
   return fact;
}
```

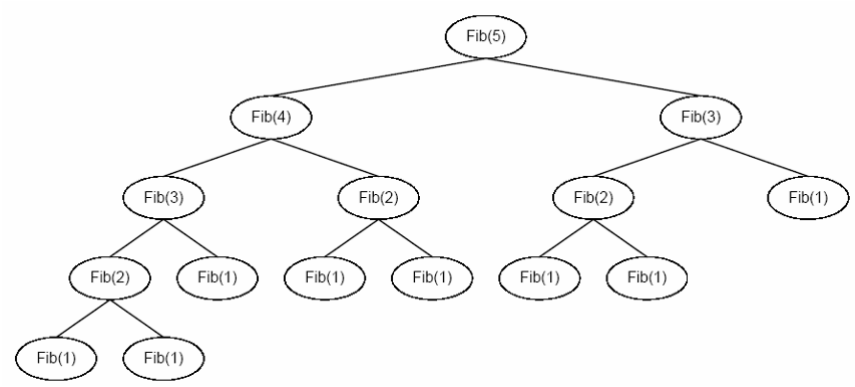

Ejemplo Práctico 3

Calcular la complejidad de un algoritmo para calcular la serie de fibonacci

```
long fibonacci_multiple(long N) {
      if (N <= 1)
         return 1;
      else
         return fibonacci_multiple(N - 1) +
                                fibonacci_multiple(N - 2);
   }

void main(String[] args) {
      long F1;
      F1 = fibonacci_multiple(10);
      System.out.println(F1);
}
```

Ejemplo Práctico 3



Ejemplo 3

- $f(n) = 1 + f(n-1) + f(n-2) \quad \forall\ n \geq 2$
- Puede observarse que $f(n)$ es una función creciente a partir de $n \geq 2$ y entonces, para esos valores, se satisface:
- $f(n) = 1 + f(n-1) + f(n-2) > 2\, f(n-2)$
- y aplicando esta misma propiedad a *f(n-2) y sucesivos f(n-i)* deberíamos tener que:
- $f(n) > 2 \cdot 2 \cdot f(n-4) > 2 \cdot 2 \cdot 2 \cdot f(n-6) > \ldots > 2^{i}\, f(n-2i) > 2^{[n/2]}$
- por inducción *n*, $f(n) > 2^{[n/2]}$ para todo $n \geq 2$.

Tiempo de cálculo

- Asumiendo que cada término se puede calcular en 1 nano segundo (10^{-9} segundos)

N	Términos calculados	Tiempo
50	33.554.432	33 μs
60	$1{,}1 \times 10^{9}$	1 seg
70	$3{,}4 \times 10^{10}$	34 seg
80	$1{,}1 \times 10^{12}$	18 min
90	$3{,}5 \times 10^{13}$	9 horas
100	$1{,}1 \times 10^{15}$	13 días
110	$1{,}4 \times 10^{17}$	4 años
120	$1{,}2 \times 10^{18}$	36 años
130	$3{,}6 \times 10^{19}$	11 siglos

Ejemplo Práctico 4

Calcular la complejidad de un algoritmo para calcular la serie de fibonacci

```
public class fibonacci {
    public static long
fibonacci_iterativo(long N) {
      if (N <= 1) {
         return 1;
      } else {
         int n0 = 1, n1 = 1, n2 = 0;
         for (int i = 1; i < N; i++) {
            n2 = n1 + n0;
            n0 = n1;
            n1 = n2;
         }
         return n2;
      }
   }
   public static void main(String[] args) {
      long F2;
      F2 = fibonacci_iterativo(10);
      System.out.println(F2);
   }
}
```

Ejemplo Práctico 5

- Dado un array A de n números
- Calcular otro array B, tal que:

$$B[i]=\frac{\sum_{j=0}^{i} A[j]}{i+1}$$

A partir de una secuencia de números A[i] calculamos otra tal que cada uno de sus elementos sea promedio de todos los anteriores en la secuencia original

* Muy usado en estadística y economía

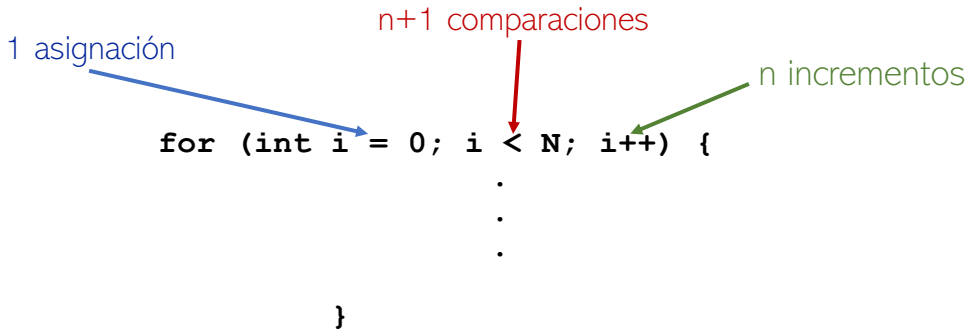
23

Ejemplo Práctico 5

```
for i=0 to n-1 do
     a=0
     for j=0 to i do
          a=a+A[j]
     B[i]=a/(i+1)
return B
```

- Partes:
  - –Inicializar y devolver array B: O(n)
  - –Bucle i: se ejecuta n veces
  - –Bucle j: se ejecuta 1+2+3+...+n=n(n+1)/2 veces: $O(n^2)$
- Total: $O(n)+O(n)+O(n^2)= O(n^2)$ (Orden cuadrático)

Ejemplo Práctico 6

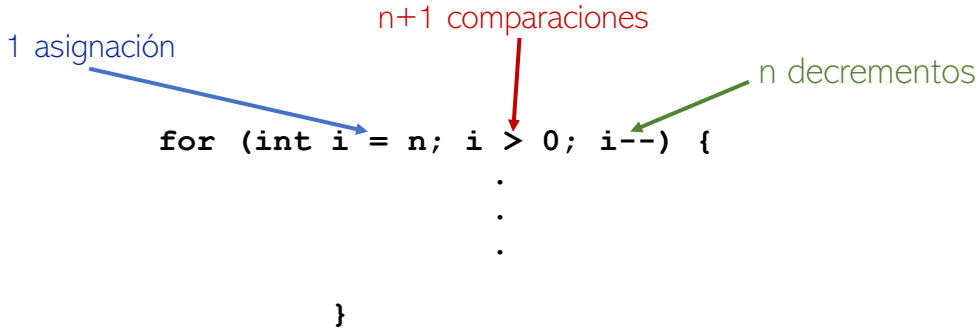


1 asignación + n+1 comparaciones + n incrementos =

1 + n + 1 + n = 2 + 2n = **2n + 2**

Complejidad de **for** básico sin modificaciones

Ejemplo Práctico 7

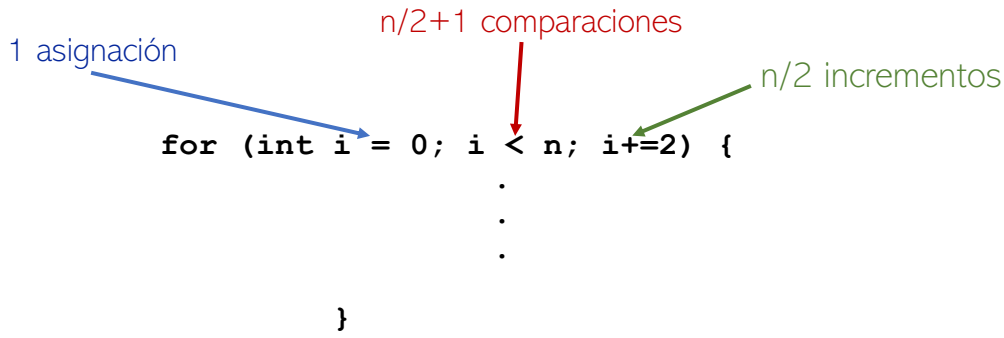


1 asignación + n+1 comparaciones + n decrementos =

1 + n + 1 + n = 2 + 2n = **2n + 2**

Complejidad

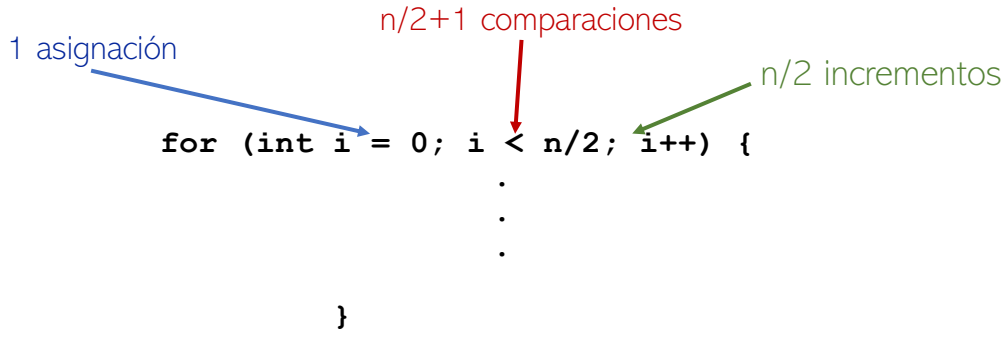
Ejemplo Práctico 8



1 asignación + n/2+1 comparaciones + n/2 incrementos =
1 + n/2 + 1 + n/2 = 2 + 2n = 2n/2 + 2

Complejidad: n+2

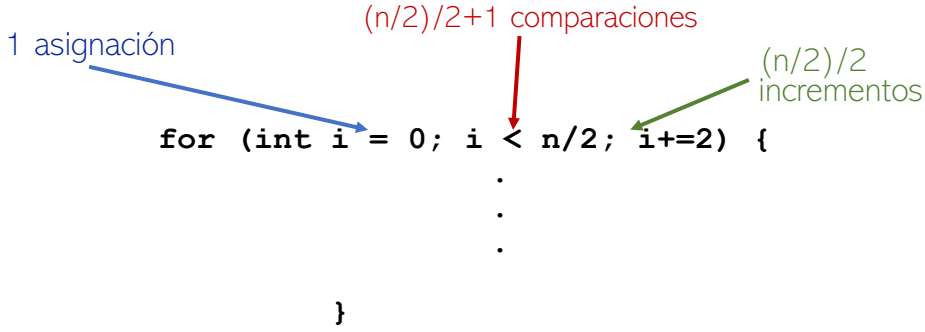
Ejemplo Práctico 9



1 asignación + n/2+1 comparaciones + n/2 incrementos =
1 + n/2 + 1 + n/2 = 2 + 2n = 2n/2 + 2

Complejidad: n+2

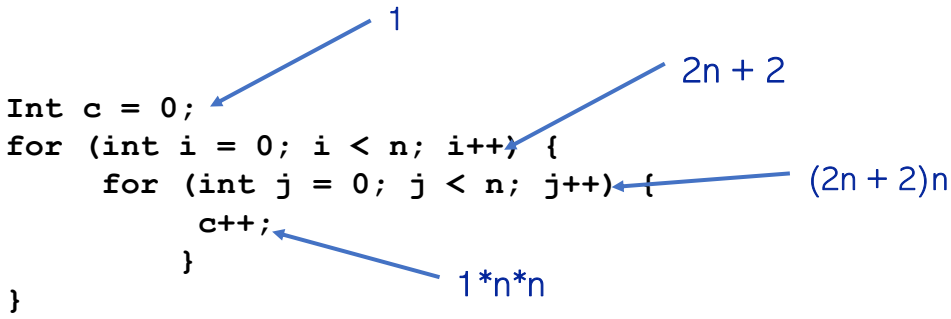
Ejemplo Práctico 10



1 asignación + (n/2)/2+1 comparaciones + (n/2)/2 incrementos =
1 + n/4 + 1 + n/4 = 2 + 2n/4 = n/2 + 2

Complejidad: n/2+2

Ejemplo Práctico 10



$1 + 2n + 2 + (2n+2)n + 1*n*n =$
$3 + 2n + 2n^2 + 2n + 2n^2 = 3n^2 + 4n + 3$
$3n^2 + 4n + 3 \rightarrow n^2 + n$

$Complejidad: n^2$

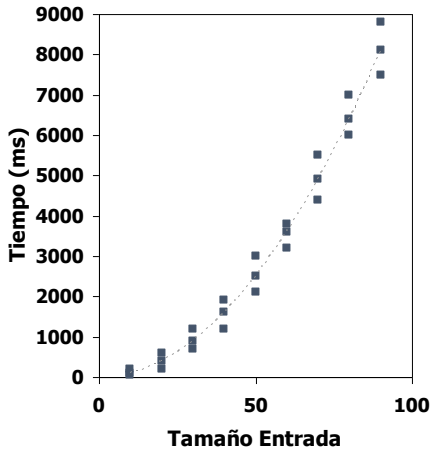
COMPLEJIDAD ASINTOTICA

Enfoques

- Enfoques para medir la complejidad de un algoritmo:
  - Enfoque empíricos o a posteriori
  - Enfoque teórico o a priori.

Complejidad computacional y asintótica

- Enfoque empíricos o *a posteriori*
  1. Escribir un programa que implemente el algoritmo, por ejemplo en C++.
  2. Ejecutar el programa con entradas de tamaño y composición variadas
  3. Usar un método como std::chrono::system_clock::now() para obtener una medida exacta del tiempo de ejecución real
  4. Trazar los resultados



Enfoque teóricos o a priori

1. Utilizar una descripción de alto nivel del algoritmo (v.g. en pseudocódigo)
2. Determinar, matemáticamente, la cantidad de recursos necesarios para ejecutar el algoritmo
3. Obtener una función genérica $f(n)$ que permita hacer predicciones sobre la utilización de recursos, siendo n el tamaño del problema

Complejidad computacional y asintótica

- Ventajas e inconvenientes
  1. El estudio *a posteriori* requiere la implementación del algoritmo (mayor coste). El estudio *a priori* solo requiere de una descripción de alto nivel (pseudocódigo).
  2. En el estudio *a posteriori* no tenemos la seguridad de los recursos que realmente se van a consumir si varían las entradas. El estudio *a priori* es independiente de los datos de entrada.
  3. En el estudio *a posteriori*, los resultados sólo serán válidos para unas determinadas condiciones de ejecución. Es difícil extrapolar los resultados si se producen cambios en el hardware, sistema operativo, lenguaje utilizado, etc. El estudio *a priori* es independiente de las condiciones de ejecución.
  4. El estudio *a posteriori* permite hacer una evaluación experimental de los recursos consumidos que no es posible en el estudio *a priori*.

Principio de invarianza

- Dos implementaciones de un mismo algoritmo no diferirán más que en una constante multiplicativa.
- Si $f_1(n)$ y $f_2(n)$ son los tiempos consumidos por dos implementaciones de un mismo algoritmo, se verifica que:

  $\exists\ c, d \in \mathbb{R}$,

  $f_1(n) \leq c\, f_2(n)$

  $f_2(n) \leq d\, f_1(n)$
- El estudio a posteriori trata con programas. El estudio a priori trata con algoritmos

$\Downarrow$

Es preferible el estudio a priori

Complejidad asintótica

- Consiste en el cálculo de la complejidad temporal a priori de un algoritmo en función del tamaño del problema, *n,* prescindiendo de factores constantes multiplicativos y suponiendo valores de n muy grandes
- No sirve para establecer el tiempo exacto de ejecución, sino que permite especificar una cota (inferior, superior o ambas) para el tiempo de ejecución de un algoritmo

Ejemplos de complejidad asintótica

- Ejemplo en pseudocódigo:

  Buscar el máximo valor de un array

```
maximoArray(A)→máximo
  Entrada A array de n enteros
  Salida elemento máximo de A
INICIO
  maxActual← A[0]
  PARA i ← 1 HASTA n – 1 HACER
    SI A[i] > maxActual
      ENTONCES maxActual ← A[i]
  DEVOLVER maxActual
FIN
```

Ejemplos de complejidad asintótica

- Ejemplo en pseudocódigo: Operaciones primitivas
  - Cómputos básicos realizados por un algoritmo
  - Identificables en pseudocódigo
  - Muy independiente del lenguaje de programación
  - La definición exacta no es importante
- Ejemplos:
  - Evaluar una expresión
  - Asignar un valor a una variable
  - Indexar un array
  - Llamar a un método
  - Retornar un valor

Ejemplos de complejidad asintótica

- Revisando el pseudocódigo, se puede estipular el número máximo de operaciones primitivas ejecutadas por un algoritmo, como una función del tamaño de la entrada

```
maximoArray(A)
INICIO
                                          operaciones
    maxActual ← A[0]                          1
    PARA i ← 1 HASTA n – 1 HACER              n
        SI A[i] > maxActual                   (n – 1)
        ENTONCES maxActual ← A[i]             (n – 1)
    { incrementar contador i }                (n – 1)
    DEVOLVER maxActual
FIN
                                    Total     4n – 2
```

Ejemplos de complejidad asintótica

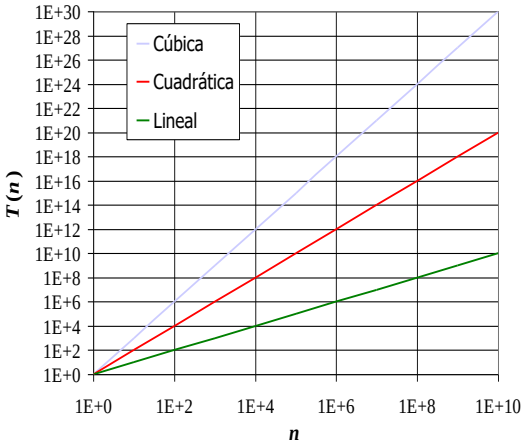
- El algoritmo *maximoArray* ejecuta $4n - 2$ operaciones primitivas en el peor de los casos
- Se definen:
  - a Tiempo tardado por la op. primitiva más rápida
  - b Tiempo tardado por la op. primitiva más lenta
- Sea $T(n)$ el tiempo de ejecución real para el peor de los casos de *maximoArray*. Entonces:
  $a(4n - 2) \leq T(n) \leq b(4n - 2)$
- Por lo tanto, el tiempo de ejecución $T(n)$ está acotado por dos funciones lineales

Ejemplos de complejidad asintótica

- Un cambio en el entorno hardware/software
  - Afecta a $T(n)$ en un factor constante, pero
  - No altera la tasa de crecimiento de $T(n)$
- La tasa de crecimiento **lineal** del tiempo de ejecución $T(n)$ es una propiedad intrínseca del algoritmo *maximoArray*

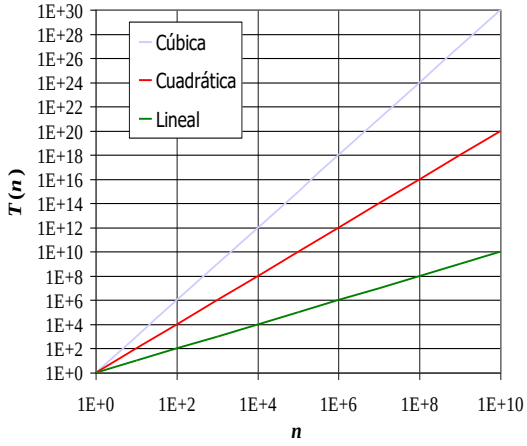
Tasa de crecimiento

- Tasa de crecimiento:
  - Lineal ≈ n
  - Cuadrática ≈ n^2
  - Cúbica ≈ n^3
- En un diagrama logarítmico, la pendiente de la línea corresponde a la tasa de crecimiento de la función



Tasa de crecimiento

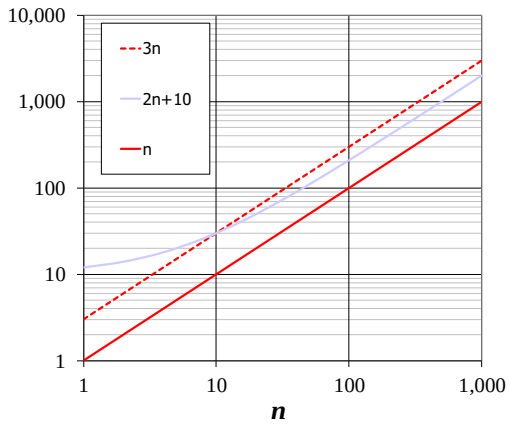
- La tasa de crecimiento no se ve afectada por:
  - factores constantes
  - términos de orden menor
- Ejemplos
  - $10^2n + 10^5$ es una función lineal
  - $10^5n^2 + 10^8n$ es una función cuadrática



La notación O

- Dadas las funciones $f(n)$ y $g(n)$, se dice que $f(n)$ es $O(g(n))$ si existen constantes positivas c y n_0 tales que

  $f(n) \leq cg(n)$ para $n \geq n_0$
- Ejemplo: $2n + 10$ es $O(n)$
  - $2n + 10 \leq cn$
  - $(c - 2)\ n \geq 10$
  - $n \geq 10/(c - 2)$
  - Elegir $c = 3$ y $n_0 = 10$



La notación O

- Ejemplo:

  $5n^3 + 3n^2 + 1 \in O(n^3)$

  tomando $n_0 = 3$, $c = 7$ tenemos que:

  $5n^3 + 3n^2 + 1 \leq 7n^3$

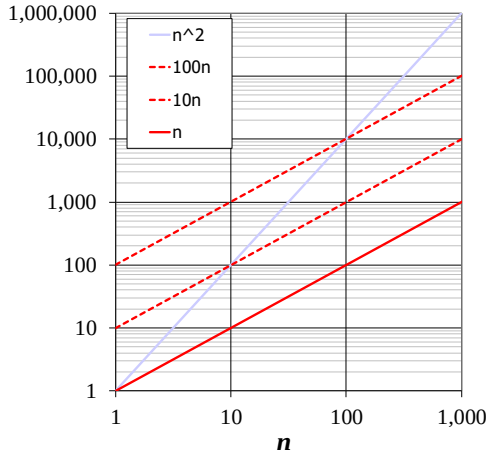
  Nota: También pertenecería a $O(n^4)$, pero no a $O(n^2)$.

La notación O

- Existen diferentes notaciones para la complejidad asintótica
- Una de ellas es la notación O, que permite especificar la cota superior de la ejecución de un algoritmo
- La sentencia "$f(n)$ es $O(g(n))$" significa que la tasa de crecimiento de $f(n)$ no es mayor que la tasa de crecimiento de $g(n)$
- La notación "O" sirve para clasificar las funciones de acuerdo con su tasa de crecimiento

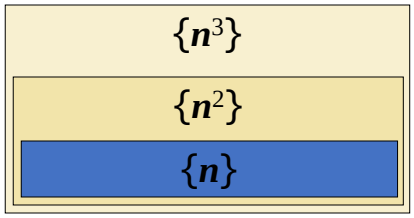
La notación O

- Ejemplo: la función n^2 no es $O(n)$
  - $n^2 \leq cn$
  - $n \leq c$
  - La desigualdad anterior no puede satisfacerse porque c debe ser una constante



La notación O

- Sea $\{g(n)\}$ la clase (conjunto) de funciones que son $O(g(n))$
- Se tiene:
  $\{n\} \subset \{n^2\} \subset \{n^3\} \subset \{n^4\} \subset \{n^5\} \subset \ldots$
  donde la inclusión es estricta



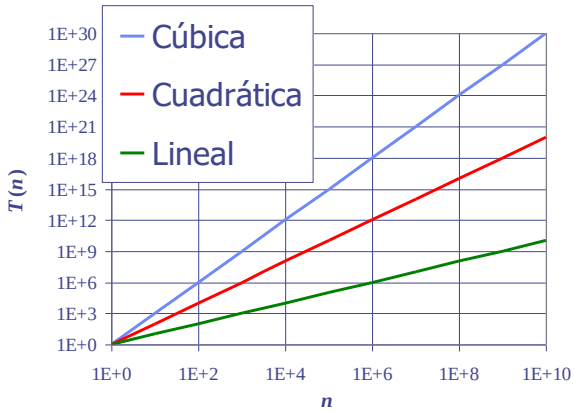
La notación O

Jerarquía de órdenes de complejidad

O(1)	Constante	No depende del tamaño del problema	Eficiente
O(log n)	Logarítmica	Búsqueda binaria	
O(n)	Lineal	Búsqueda lineal	
O(n•log n)	Casi lineal	Quick-sort	
$O(n^2)$	Cuadrática	Algoritmo de la burbuja	Tratable
$O(n^3)$	Cúbica	Producto de matrices	
$O(n^k)$ k>3	Polinómica		
$O(k^n)$ k>1	Exponencial	Algunos algoritmos de grafos	Intratable
O(n!)	Factorial		

La notación O

Jerarquía de órdenes de complejidad



La notación O

- Cambios en el entorno HW o SW afectan a factores constantes (principio de invarianza) pero no al orden de complejidad O(f(n))
- Hay que buscar algoritmos con el menor orden de complejidad
- La eficiencia es un término relativo que depende del problema
- El análisis de la eficiencia es asintótico → sólo es válido para tamaños de problema suficientemente grandes

La notación O

- El análisis asintótico de algoritmos determina el tiempo de ejecución en notación "O"
- Para realizar el análisis asintótico
  - Buscar el número de operaciones primitivas ejecutadas en el peor de los casos como una función del tamaño de la entrada
  - Expresar esta función con la notación "O"
- Ejemplo:
  - Se sabe que el algoritmo ***maximoArray*** ejecuta como mucho $4n - 2$ operaciones primitivas
  - Se dice que ***maximoArray*** "ejecuta en un tiempo $O(n)$"
- Como se prescinde de los factores constantes y de los términos de orden menor, se puede hacer caso omiso de ellos al contar las operaciones primitivas

La notación O

- Reglas:
  - Si $f(n)$ es un polinomio de grado d, entonces $f(n)$ es $O(n^d)$, es decir,
    1. Prescindir de los términos de orden menor
    2. Prescindir de los factores constantes
  - Usar la clase más pequeña posible de funciones
    - Decir "$2n$ es $O(n)$" en vez de "$2n$ es $O(n^2)$"
  - Usar la expresión más simple para la clase
    - Decir "$3n + 5$ es $O(n)$" en vez de "$3n + 5$ es $O(3n)$"

La notación O

- Reglas prácticas:
  - Operaciones **primitivas**: O(1)
  - **Secuencia** de instrucciones: máximo de la complejidad de cada instrucción (regla de la suma)
  - **Condiciones simples**: Tiempo necesario para evaluar la condición más el requerido para ejecutar la consecuencia (peor caso).
  - **Condiciones alternativas**: Tiempo necesario para evaluar la condición más el requerido para ejecutar el mayor de los tiempos de las consecuencias (peor caso).

La notación O

- Reglas prácticas:
  - **Bucle con iteraciones fijas**: multiplicar el número de iteraciones por la complejidad del cuerpo (regla del producto).
  - **Bucle con iteraciones variables**: Igual pero poniéndose en el peor caso (ejecutar el mayor número de iteraciones posible).
  - **Llamadas a subprogramas**, funciones o métodos: Tiempo de evaluación de cada parámetro más el tiempo de ejecución del cuerpo.

Medias prefijas (Cuadrático)

```
mediasPrefijas1 (vector,n) → vector
INICIO
    A ← nuevo Array de n enteros
    i ← 0
    MIENTRAS (i < n)
        s ← X[0]
        j ← 1
        MIENTRAS (j <= i)
            s ← s + X[j]
            j ← j + 1
        FIN-MIENTRAS
        A[i] ← s / (i + 1)
        i ← i + 1
    FIN-MIENTRAS
    DEVOLVER A
FIN
```

- Tiempo de ejecución = $O(n^2)$

Medias prefijas (Lineal)

```
mediasPrefijas2 (vector,n) → vector
INICIO
    A ← nuevo Array de n enteros
    s ← 0
    i ← 0
    MIENTRAS (i < n)
        s ← s + X[j]
        A[i] ← s / (i + 1)
        i ← i + 1
    FIN-MIENTRAS
    DEVOLVER A
FIN
```

- Tiempo de ejecución = $O(n)$

Análisis del caso mejor, peor y medio

El tiempo de ejecución de un algoritmo puede variar con los datos de entrada.

Ejemplo (ordenación por inserción): :

```
INICIO
        i ← 1
        MIENTRAS (i < n)
                x ← T[i]
                j ← i - 1
                MIENTRAS (j > 0 Y T[j] > x)
                        T[j+1] ← T[j]
                        j ← j - 1
                FIN-MIENTRAS
                T[j+1] ← x
                i ← i + 1
        FIN-MIENTRAS
FIN
```


Análisis del caso mejor

Si el vector está completamente ordenado (mejor caso) → el segundo bucle no realiza ninguna iteración→ Complejidad O(n).

Análisis del peor mejor

- Si el vector está ordenado de forma decreciente, el peor caso se produce cuando x es menor que T[j] para todo j entre 1 e i -1.
  - En esta situación la salida del bucle se produce cuando j = 0.
  - En este caso, el algoritmo realiza i – 1 comparaciones.
  - Esto será cierto para todo valor de i entre 1 y n -1 si el orden es decreciente.
  - El número total de comparaciones será:

$$\sum_{i=1}^{n-1}(i-1)=\frac{n\bullet(n-1)}{2}\in O(n^2)$$

Análisis del caso promedio

- La complejidad en el caso promedio estará entre O(n) y $O(n^2)$.
  - Para hallarla habría que calcular matemáticamente la complejidad de todas las permutaciones de todos valores que se pueden almacenar en el vector.
  - Se puede demostrar que es $O(n^2)$.

ORDEN DE MAGNITUD

Eficiencia (Ordenes de Magnitud)

- *Bucles Lineales:*
  - $E = F(n) = n$

```
i=0
Mientras i<n
        Código
        i++
Fin M
```

- *Bucles logarítmicos: avance i*=a*
  - $E = F(n) = Log_2\ n$
  - $E = F(n) = Log_a\ n$

```
i=1
Mientras i<n
        Código
        i*=2
Fin M
```

- *Bucles Anidados: para "y" bucles de "n" iteraciones c/u*
  - $E = F(n) = n_1 * n_2 * \ldots . n_y = n^y$
- *Recursividad: se invoca "K" veces a si mismo con un lote inicial "n"*
  - $E = F(n) = k^n$

Notación O()

- 7n-3 es O(n)
  - c=7, n0=1
  - 7n-3 ≤ 7n
- $20n^3 + 10N \log(n) + 5$ es O(n3)
- $3\log(n) + \log(\log(n))$ es $O(\log(n))$
- 2^{100} es O(1)
- 5/n es O(1/n)

- $\log n \in O(n^{\alpha}),\ \alpha > 0$
- $n^k \in O(c^n),\ c > 1$

Ejercicio 6

- Ordenar las siguientes funciones con respecto de su orden de crecimiento.
  - $\log n$
  - $4^{\log n}$
  - 2^n
  - 2^2
  - 1/n
  - $(3/2)^n$

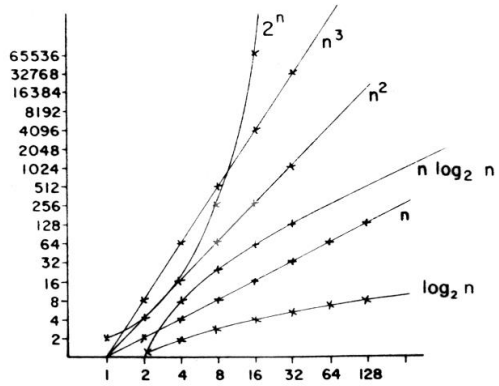
Ejercicio 7

- Ordenar las siguientes funciones con respecto de su orden de crecimiento.

$2^{(\log n)/2}$	$\log(\log n)$	n	n^3
e^n	$n!$	$n^{\log(\log n)}$	$n \log n$
$(3/2)^n$	$2^{\log n}$	n^2	$n\,2^n$
2^n	2^2	$4^{\log n}$	$(n+1)!$
$n^{1/\log n}$	$(\log n)^{1/2}$	$\log n$	2^{2n+1}

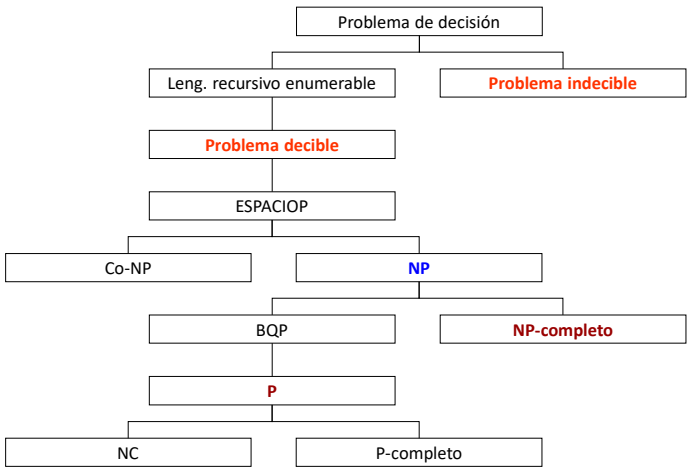
Orden de magnitud de un algoritmo

- Tiempos más comunes de los algoritmos:
- O(1) < O(log n) < O(n) < O(n log n) < O(n2) < O(n3) < O(2n)

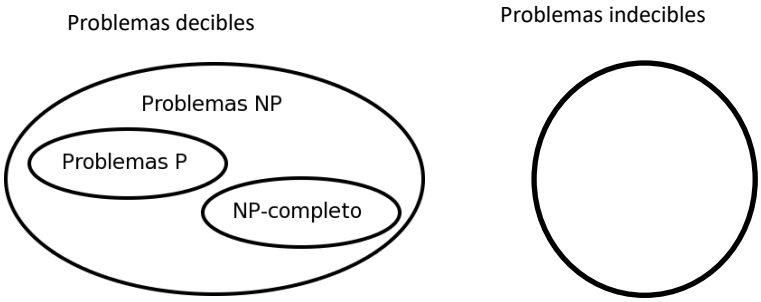


CLASES DE PROBLEMAS

Problema de Decisión



Clases de Problemas



73

Problema de Decisión

- Es un conjunto de frases de longitud finita que tienen asociadas frases resultantes también de longitud finita.
- Un problema de decisión es un problema en donde las respuestas posibles son SI o NO.
- Un problema de decisión también se puede formalizar como el problema de decidir si una cierta frase pertenece a un conjunto dado de frases, también llamado lenguaje formal.
- El conjunto contiene exactamente las frases para las cuales la respuesta a la pregunta es positiva.

¿Cierto número entero es primo?
Una instancia de este problema sería: ¿17 es primo?.

Ejemplo de problemas

- Las frases sobre el alfabeto {a, b} que contienen alternadas las letras a y b.
- Las frases sobre el alfabeto {a, b, c} que contienen igual número de letras a y b.
- Las frases que describen un grafo con aristas etiquetadas con números naturales que indican su longitud, dos vértices del grafo y un camino en el grafo que es el camino más corto entre esos dos vértices.

todos los problemas matemáticos pueden ser redactados para que tomen la forma de un problema de decisión. Las soluciones al problema de decisión y al problema original se diferencian a lo sumo por un factor lineal.

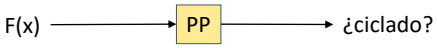
1. Problema Decible

- Si existe un algoritmo que pueda decidir para cada posible frase de entrada si esa frase pertenece al lenguaje, entonces se dice que el problema es decidible, de otra forma se dice que es un problema indecidible.
- Se estudia cuanto recurso necesita un algoritmo decidible para ejecutarse (tiempo, espacio, procesadores, máquina)
- Estos pueden ser:
  - Problemas intratables No se pueden resolver en tiempo polinómico $O(k^n)$
  - Problemas tratables Se resuelven con algoritmos polinómicos $O(n^k)$

2. Problemas indecibles

- O "problemas no solucionables en forma algorítmica".
- Son problemas que se pueden describir, pero no se pueden representar o resolver.

- Ejemplo: Problema de la parada.
  - No existe ningún algoritmo que diga si un programa va a parar en tiempo finito con una entrada dada:



3. Problemas NP

- Su nombre viene de *Polinómico no determinista* (*Non-Deterministic Polynomial-time*).

- Es el conjunto de problemas que pueden ser resueltos en tiempo polinómico por una máquina de Turing no determinista.

- La importancia de esta clase de problemas de decisión es que contiene muchos problemas de búsqueda y de optimización para los que se desea saber si existe una cierta solución o si existe una mejor solución que las conocidas.

3.1. Problemas P

- Un problema pertenece a la clase P si puede ser resuelto en tiempo polinomial en una computadora determinística.
- Ejemplos: Quicksort, búsqueda binaria, multiplicación matricial.
- Esto es si el tiempo de ejecución del algoritmo es menor que un cierto valor calculado a partir del número de variables implicadas (variables de entrada) usando una fórmula polinómica.

- Si determinar el camino óptimo que debe recorrer un cartero que pasa por N casas necesita menos de $50*N^2+N$ segundos, entonces el problema es resoluble en un "tiempo polinómico".

3.2. Problemas NP-completo

- En teoría de la complejidad computacional, la clase de complejidad NP-completo es el subconjunto de los problemas de decisión en NP tal que todo problema en NP se puede reducir en cada uno de los problemas de NP-completo.
- Se puede decir que los problemas de NP-completo son los problemas más difíciles de NP y muy probablemente no formen parte de la clase de complejidad P.
- La razón es que de tenerse una solución polinómica para un problema de NP-completo, todos los problemas de NP tendrían también una solución en tiempo polinómico.

3.2. Problemas NP-completo

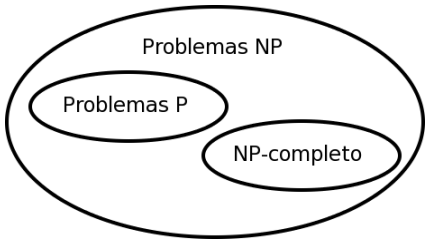
- Todos los algoritmos conocidos para problemas NP-completos utilizan tiempo exponencial con respecto al tamaño de la entrada.
- Todos los algoritmos requeridos para resolverlos requieren tiempo exponencial en el peor caso.
- Es decir, son sumamente difíciles de resolver.
- Ejemplos: el problema del agente viajero, $O(n^2 2^n)$

P vs NP

- La clase P contiene problemas que pueden resolverse rápidamente.
- La clase NP contiene problemas cuya solución puede verificarse rápidamente.
- En 1971 se planteó la pregunta: ¿Es P = NP? Desde entonces, sigue siendo una pregunta abierta para los teóricos.
- Se cree que P != NP

¿P=NP?

- Observa que si hubiera un algoritmo polinómico para cualquier problema en NP-C entonces P=NP.
- La hipótesis más aceptada es que P≠NP.
- Si queremos enfrentar un problema NPC es nejor buscar alternativas (simplificaciones, aproximaciones, etc.).



School of Computer Science

SOLUCIONES APROXIMADAS

Soluciones aproximadas

- Se desconoce si hay mejores algoritmos, por la cual, para resolver un problema NP-completo de tamaño arbitrario se utiliza uno de los siguientes enfoques:

- Aproximación:
- Probabilístico:
- Casos particulares:
- Heurísticas:
- Algoritmo genético:

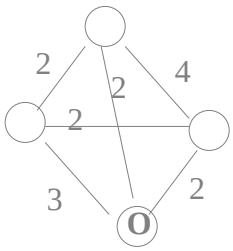
Soluciones aproximadas

- Aproximación: Un algoritmo que rápidamente encuentra una solución no necesariamente óptima, pero dentro de un cierto rango de error. En algunos casos, encontrar una buena aproximación es suficiente para resolver el problema, pero no todos los problemas NP-completos tienen buenos algoritmos de aproximación.
- Probabilístico: Un algoritmo probabilístico obtiene en promedio una buena solución al problema planteado, para una distribución de los datos de entrada dada.
- Casos particulares: Cuando se reconocen casos particulares del problema para los cuales existen soluciones rápidas.
- Heurísticas: Un algoritmo que trabaja razonablemente bien en muchos casos. En general son rápidos, pero no existe medida de la calidad de la respuesta.
- Algoritmo genético: Algoritmos que mejoran las posibles soluciones hasta encontrar una que posiblemente esté cerca del óptimo. Tampoco existe forma de garantizar la calidad de la respuesta.

Ejemplo

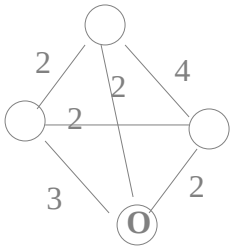
Problema (TSP)

- Sea: un conjunto de ciudades.
- Una ciudad origen O.
- Un conjunto de aristas que une las ciudades, con costos asociados.
- Problema del viajante: recorrer todas las ciudades, comenzando en O y terminando en O, al menor costo posible.

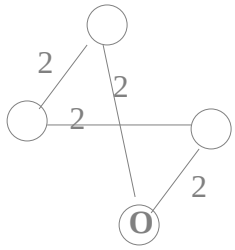


Ejemplo

- Problema



- Una solución óptima (de valor 8)



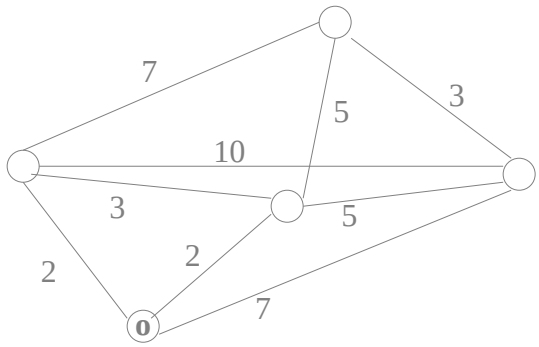
Métodos de aproximación

- Encuentra solución con error máximo conocido a priori.
- En algunos casos, error de aproximación fijo.
- En otros, posible elegir trade-off entre error de aproximación y esfuerzo computacional (mayor esfuerzo, menor error).

Método goloso (greedy)

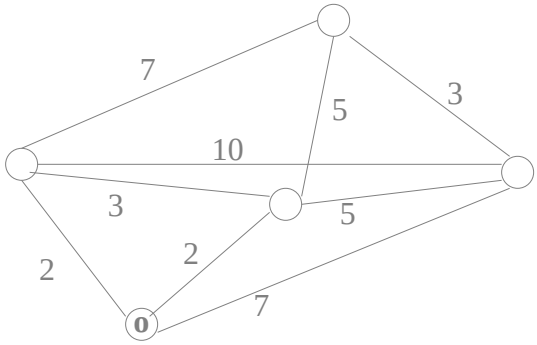
- Idea: tratar de construir una solución eligiendo de manera iterativa los elementos componentes de menor costo.
- Para algunos problemas con estructura particular, la solución construida es una solución óptima. En general, no es el caso.

Aplicación al TSP

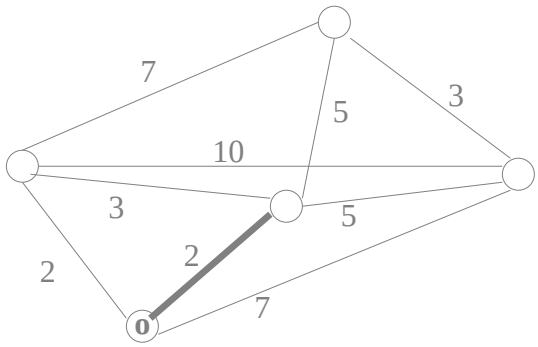


Aplicación al TSP

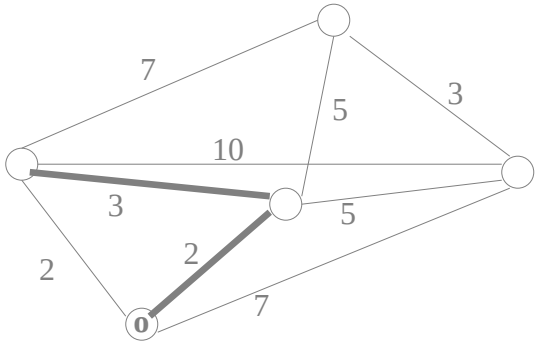
- Elijo la ciudad más próxima (menor costo) entre las aún no visitadas.



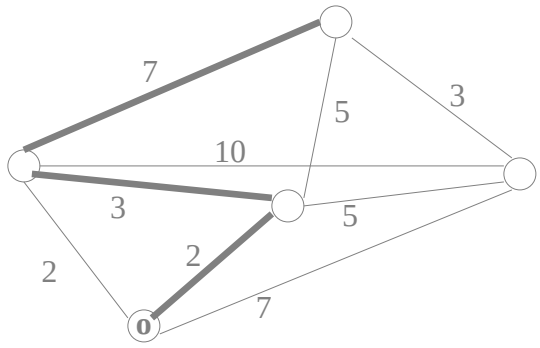
Aplicación al TSP



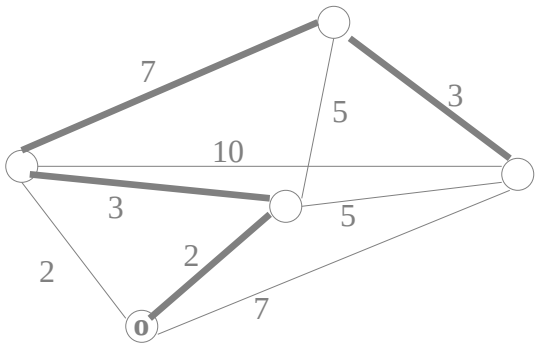
Aplicación al TSP



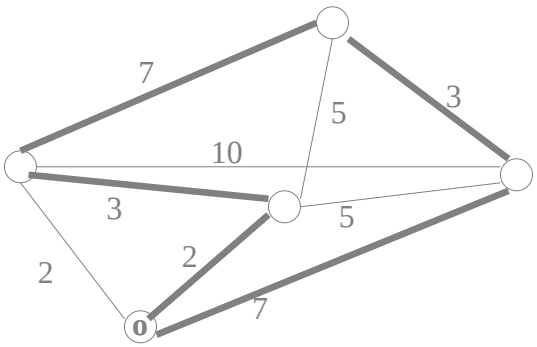
Aplicación al TSP



Aplicación al TSP

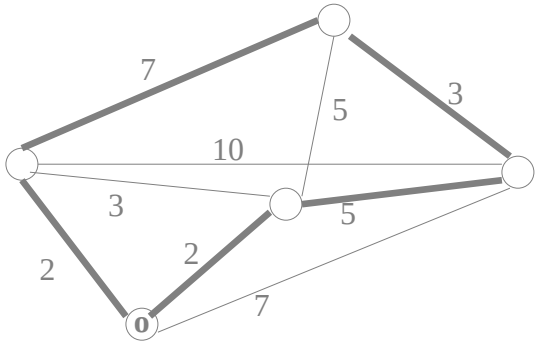


Aplicación al TSP



Solución:
costo 22

Aplicación al TSP



Solución alternativa:
costo 19

School of Computer Science

Thanks!

Questions?

M.Sc. Franci Suni Lopez
fsunilo@unsa.edu.pe