Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencias de la Computación

Clase 13: Algoritmos de Ordenación

Rodrigo Toro Icarte (rntoro@uc.cl)

IIC1103 Introducción a la Programación - Sección 5

04 de Mayo, 2015

Clases pasadas

El mundo está lleno de listas.

Apellido Paterno	Apellido Materno	Nombre
ÁLVAREZ	JOHNSON	SOFÍA PAZ
ANDRADE	POBLETE	ISIDORA CAROLINA
AYLWIN	REYES	MANUEL EDUARDO
BALBONTÍN	GUMUCIO	NICOLÁS FELIPE
BRAHM	ESCALONA	THOMAS IGNACIO
BRAVO	DE LA CRUZ	JUAN PABLO
BUSTAMANTE	ALONZO	YAEL FERNANDA
CABROLIER	OSSES	BIANCA MACARENA
CÁDIZ	BOSCH	CRISTIAN RICARDO
CANALES	CÓRDOVA	FRANCISCA MACARENA
CANIO	FERNÁNDEZ	NELSON RUBÉN
CARRASCO	SALINAS	RICARDO ANDRÉS
CARVAJAL	MEJIA	NICOLAS IGNACIO
CASTILLO	FUENTES	PABLO FRANCISCO

Clases pasadas

El mundo está lleno de listas.

Clases pasadas

Sintaxis

lista = [elemento_1, elemento_2, elemento_3, ...]

Indexable: Sus elementos se obtienen indicando su índice.

```
1 l = [3,4,2,4,9,6]   # lista de números
2 print(l[3])         # >>> 4
3 print(l[-2])        # >>> 9
```

Mutable: Sus elementos pueden ser modificados.

```
1 l = [3,4,2,4,9,6]   # lista de números
2 l[0] = 6
3 print(l)            # >>> [6,4,2,4,9,6]
```

Clase pasada

Para recorrer una lista l podemos usar...

... while

```
i = 0
while(i < len(l)):
  print(l[i])
  i += 1
```

... for

```
# itero sobre índices
for i in range(len(l)):
  print(l[i])

# itero sobre elementos
for e in l:
  print(e)
```

Complejidad Computacional

Complejidad Computacional

¿Existen programas más lentos que otros?

Complejidad Computacional

¿Existen programas más lentos que otros? ¿Por qué?

Complejidad Computacional

¿Existen programas más lentos que otros? ¿Por qué?

Complejidad: La complejidad de un algoritmo es una medida de cuántas instrucciones necesita para resolver un problema.

Complejidad Computacional

¿Existen programas más lentos que otros? ¿Por qué?

Complejidad: La complejidad de un algoritmo es una medida de cuántas instrucciones necesita para resolver un problema.

¿Cuándo un algoritmo es más *complejo* que otro?

Complejidad Computacional

Ejemplo:

```
1 def es_lista_ordenada(l):
2   for i in range(len(l) - 1):
3     if(l[i] > l[i+1]):
4       return False
5   return True
```

¿Qué hace este algoritmo?

Complejidad Computacional

Ejemplo:

```
1 def es_lista_ordenada(l):
2   for i in range(len(l) - 1):
3     if(l[i] > l[i+1]):
4       return False
5   return True
```

¿Qué hace este algoritmo?

¿Cuántas instrucciones ejecuta aproximadamente?

Complejidad Computacional

La complejidad se evalúa para el peor caso.

Complejidad Computacional

La complejidad se evalúa para el peor caso.

```
1 def es_lista_ordenada(l):
2   for i in range(len(l) - 1):
3     if(l[i] > l[i+1]):
4       return False
5   return True
```

¿Cuál es el peor caso en este ejemplo?

Complejidad Computacional

La complejidad se evalúa para el peor caso.

```
1 def es_lista_ordenada(l):
2   for i in range(len(l) - 1):
3     if(l[i] > l[i+1]):
4       return False
5   return True
```

¿Cuál es el peor caso en este ejemplo?

¿Cuántas instrucciones ejecuta aproximadamente?

Complejidad Computacional

Para definir la complejidad de un algoritmo se usa la notación $O(.)$

Complejidad Computacional

Para definir la complejidad de un algoritmo se usa la notación $O(.)$

Por lo general se expresan en función de:

- constantes (k).
- variables (n).

Ejemplos: $O(k)$, $O(k \cdot n)$, $O(k(\cdot n^2 + n + 1))$, etc...

Complejidad Computacional

Para definir la complejidad de un algoritmo se usa la notación $O(.)$

Por lo general se expresan en función de:

- constantes (k).
- variables (n).

Ejemplos: $O(k)$, $O(k \cdot n)$, $O(k(\cdot n^2 + n + 1))$, etc...

Idea: A medida que n crece, la complejidad queda dominada por la variable que crece más rápido.

Complejidad Computacional

Sin importar el valor de k_i, siempre existe un valor de n a partir del cual:

$$k_1 < k_2 \cdot n < k_3 \cdot n^2$$

[1] porque no son informativas...

Complejidad Computacional

Sin importar el valor de k_i, siempre existe un valor de n a partir del cual:

$$k_1 < k_2 \cdot n < k_3 \cdot n^2$$

... luego para definir complejidad podemos quitar las constantes[1]:

- $O(k) = O(1)$
- $O(k \cdot n) = O(n)$
- $O(k \cdot n^2) = O(n^2)$
- ...

[1] porque no son informativas...

Complejidad Computacional

Sin importar el valor de n, siempre existe un valor de n a partir del cual:

$$100000 \cdot n^2 + 100 \cdot n < n^3$$

Complejidad Computacional

Sin importar el valor de n, siempre existe un valor de n a partir del cual:

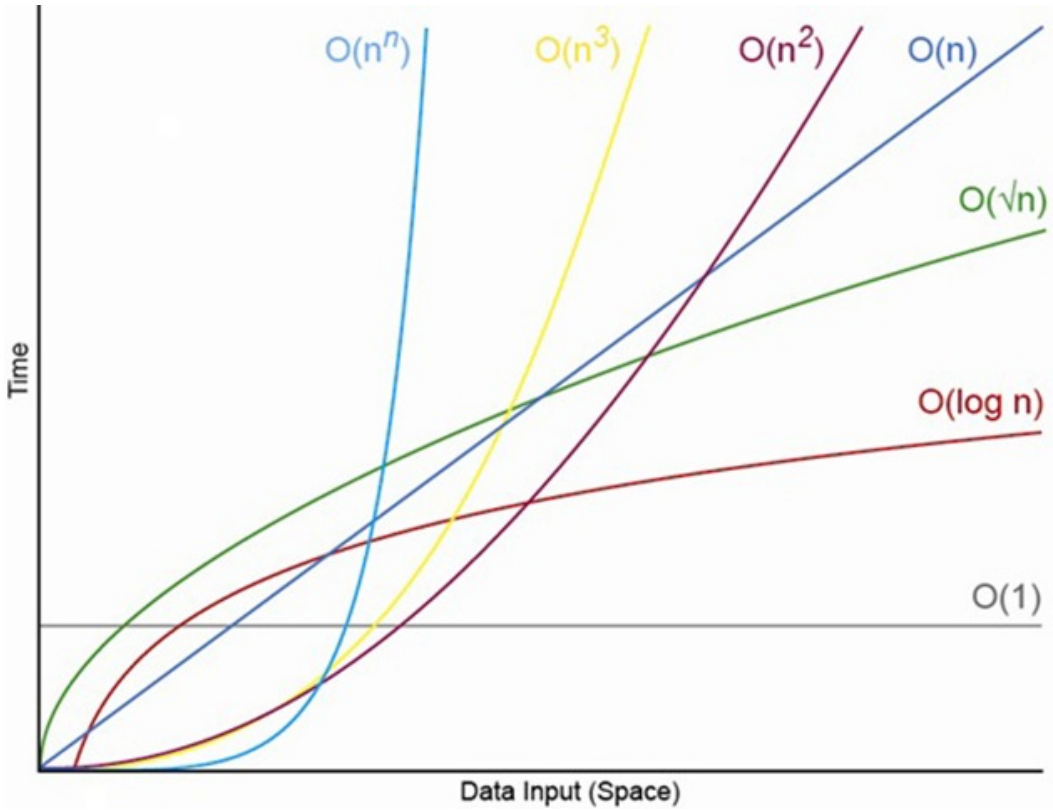
$$100000 \cdot n^2 + 100 \cdot n < n^3$$

... luego nos interesa solo el término que crezca más rápido:

- $O(n^2 + n + 1) = O(n^2)$
- $O(n^4 + 80 \cdot n^2 + 10000) = O(n^4)$
- $O(n^5 + 2^n) = O(2^n)$

Complejidad Computacional

Algunas complejidades comunes:



Complejidad Computacional

Ejemplo 1:

Encontrar el máximo en una lista desordenada

```
1 def get_max(l):
2   m = -float('inf')
3   for e in l:
4     if(m < e):
5       m = e
6   return m
```

Complejidad Computacional

Ejemplo 1:

Encontrar el máximo en una lista desordenada

```
1 def get_max(l):
2   m = -float('inf')
3   for e in l:
4     if(m < e):
5       m = e
6   return m
```

... ejecuta aproximadamente $k \cdot n$ instrucciones (con k constante y $n = len(l)$) $\approx O(n)$.

Complejidad Computacional

Ejemplo 2:

Encontrar el máximo en una lista ordenada

```
1 def get_max_sorted(l):
2   return l[len(l) - 1]
```

Complejidad Computacional

Ejemplo 2:

Encontrar el máximo en una lista ordenada

```
1 def get_max_sorted(l):
2   return l[len(l) - 1]
```

... ejecuta aproximadamente k instrucciones (con k constante) $\approx O(1)$..

Complejidad Computacional

A medida que *len(l)* crece, la diferencia en tiempo de ejecución entre ambos métodos aumenta.

len(l)	speedup ($O(n)/O(1)$)
10	3,5
1.000	36,2
100.000	3.439,9
10.000.000	338.460,7

Complejidad Computacional

A medida que *len(l)* crece, la diferencia en tiempo de ejecución entre ambos métodos aumenta.

len(l)	speedup ($O(n)/O(1)$)
10	3,5
1.000	36,2
100.000	3.439,9
10.000.000	338.460,7

... ciertamente obtener el máximo en una lista ordenada es más rápido.

Algoritmos de ordenación

Problema: Queremos ordenar los elementos de una lista.

Algoritmos de ordenación

Problema: Queremos ordenar los elementos de una lista.

Ejemplo: Ordenar lista de estudiantes según PPA.

Algoritmos de ordenación

Problema: Queremos ordenar los elementos de una lista.

Ejemplo: Ordenar lista de estudiantes según PPA.

¿Por qué ordenar es importante?

Algoritmos de ordenación

Problema: Queremos ordenar los elementos de una lista.

Ejemplo: Ordenar lista de estudiantes según PPA.

¿Por qué ordenar es importante?

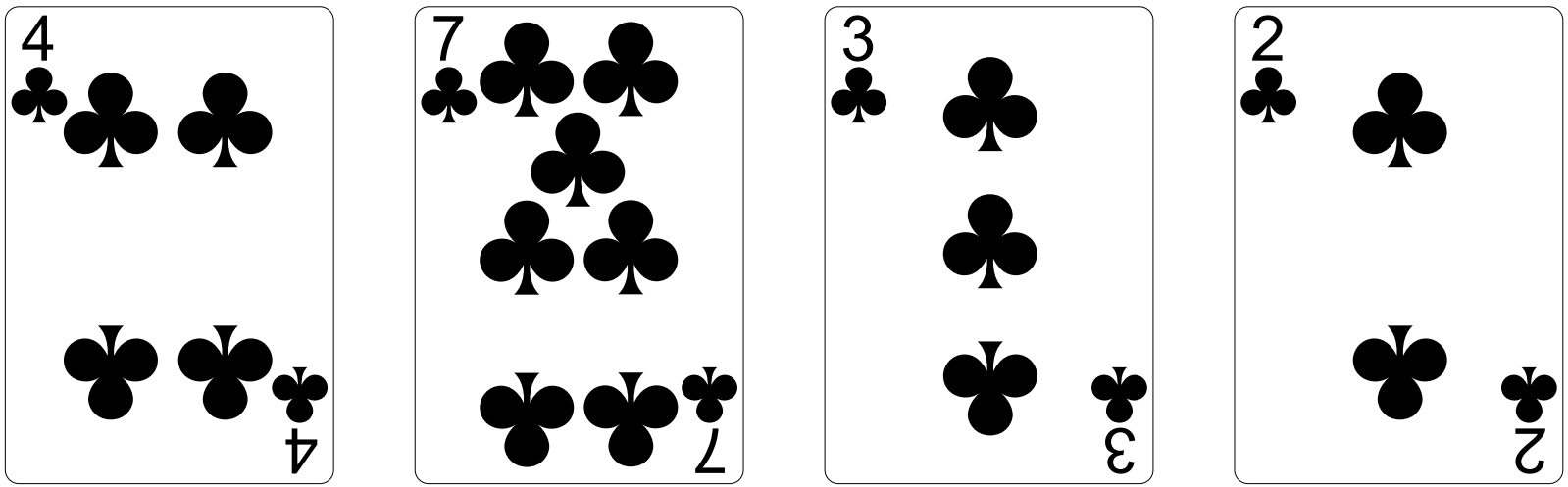
- Nos da una nueva visión de los datos.
- Permite obtener máximo, mínimo y percentiles en forma eficiente.
- Permite encontrar elementos en una lista más rápido.

Algoritmos de ordenación

Ejemplo

Algoritmos de ordenación

¿Cómo ordenarían un mazo de cartas?



Algoritmos de ordenación: SelectSort

Idea: En cada paso encontramos el elemento mínimo y lo dejamos al comienzo de la lista.

Algoritmos de ordenación: SelectSort

Idea: En cada paso encontramos el elemento mínimo y lo dejamos al comienzo de la lista.

En más detalle: Recorrer cada posición de la lista **l** (desde **0** hasta **len(l)-1**)

1. Sea **i** la posición actual
2. Sea **j** el índice del elemento mínimo en **l[i:]**
3. Intercambiar **i** por **j**.

Algoritmos de ordenación: SelectSort

Ejemplo: $l = [4, 7, 3, 2]$

Algoritmos de ordenación: SelectSort

Ejemplo: l = [4, 7, 3, 2]

Analizo i = 0: 4 7 3 2 → el min(l[i:]) es j = 3
Analizo i = 0: **4** 7 3 **2** → intercambio l[i] con l[j]

Algoritmos de ordenación: SelectSort

Ejemplo: l = [4, 7, 3, 2]

Analizo i = 0: 4 7 3 2 → el min(l[i:]) es j = 3
Analizo i = 0: **4** 7 3 **2** → intercambio l[i] con l[j]

Analizo i = 1: 2 7 3 4 → el min(l[i:]) es j = 2
Analizo i = 1: 2 **7** **3** 4 → intercambio l[i] con l[j]

Algoritmos de ordenación: SelectSort

Ejemplo: l = [4, 7, 3, 2]

Analizo i = 0: 4 7 3 2 → el min(l[i:]) es j = 3
Analizo i = 0: **4** 7 3 **2** → intercambio l[i] con l[j]

Analizo i = 1: 2 7 3 4 → el min(l[i:]) es j = 2
Analizo i = 1: 2 **7** **3** 4 → intercambio l[i] con l[j]

Analizo i = 2: 2 3 7 4 → el min(l[i:]) es j = 3
Analizo i = 2: 2 3 **7** **4** → intercambio l[i] con l[j]

Algoritmos de ordenación: SelectSort

Ejemplo: l = [4, 7, 3, 2]

Analizo i = 0: 4 7 3 2 → el min(l[i:]) es j = 3
Analizo i = 0: **4** 7 3 **2** → intercambio l[i] con l[j]

Analizo i = 1: 2 7 3 4 → el min(l[i:]) es j = 2
Analizo i = 1: 2 **7** **3** 4 → intercambio l[i] con l[j]

Analizo i = 2: 2 3 7 4 → el min(l[i:]) es j = 3
Analizo i = 2: 2 3 **7** **4** → intercambio l[i] con l[j]

Fin: 2 3 4 7

Algoritmos de ordenación: SelectSort

Algoritmo:

Algoritmos de ordenación: SelectSort

Algoritmo:

```
1  def selectSort(l):
2    # Recorremos elementos de la lista
3    for i in range(len(l)-1):
4      # encuentro posición del elemento mínimo desde i
       en adelante
5      pos = i
6      for j in range(pos+1,len(l)):
7        if(l[pos] > l[j]):
8          pos = j
9      # Intercambio 'i' por posición del mínimo (pos)
10     auxiliar = l[pos]
11     l[pos] = l[i]
12     l[i] = auxiliar
13
14   # Retornamos lista ordenada
15   return l
```

Algoritmos de ordenación: SelectSort

¿Cuál es la complejidad de SelectSort?

Algoritmos de ordenación: SelectSort

¿Cuál es la complejidad de SelectSort?

Sea $n = len(l)$ y k una constante, entonces el número de instrucciones aproximado (en el peor caso) es:

$$k(n + (n-1) + (n-2) + ... + 2)$$

$$k\sum_{i=1}^{n} i - 1 = k\frac{n(n+1)}{2} - 1 \approx O(n^2)$$

Algoritmos de ordenación: InsertSort

Algoritmos de ordenación: InsertSort

Idea: Tomo uno a uno cada elemento de la lista, y me aseguro de que no exista ningún elemento menor a su izquierda.

Algoritmos de ordenación: InsertSort

Idea: Tomo uno a uno cada elemento de la lista, y me aseguro de que no exista ningún elemento menor a su izquierda.

En más detalle: Recorrer cada posición de la lista **l** (desde **1** hasta **len(l)**)

1. Sea **i** la posición actual
2. Sea **j = i - 1**
3. Si **l[i]** $<$ **l[j]** intercambio **i** por **j** y vuelvo a 1.

Algoritmos de ordenación: InsertSort

Ejemplo: $l = [4, 7, 3, 2]$

Algoritmos de ordenación: InsertSort

Ejemplo: l = [4, 7, 3, 2]

Analizo 4: **4** 7 3 2 → ok!
Analizo 7: 4 **7** 3 2 → ok!

Algoritmos de ordenación: InsertSort

Ejemplo: l = [4, 7, 3, 2]

Analizo 4: **4** 7 3 2 → ok!
Analizo 7: 4 **7** 3 2 → ok!

Analizo 3: 4 7 **3** 2 → 7 > 3 intercambio!
Analizo 3: 4 **3** 7 2 → 4 > 3 intercambio!
Analizo 3: **3** 4 7 2 → 4 > 3 ok!

Algoritmos de ordenación: InsertSort

Ejemplo: l = [4, 7, 3, 2]

Analizo 4: **4** 7 3 2 → ok!
Analizo 7: 4 **7** 3 2 → ok!

Analizo 3: 4 7 **3** 2 → 7 > 3 intercambio!
Analizo 3: 4 **3** 7 2 → 4 > 3 intercambio!
Analizo 3: **3** 4 7 2 → 4 > 3 ok!

Analizo 2: 3 4 7 **2** → 7 > 2 intercambio!
Analizo 2: 3 4 **2** 7 → 4 > 2 intercambio!
Analizo 2: 3 **2** 4 7 → 3 > 2 intercambio!
Analizo 2: **2** 3 4 7 → ok!

Algoritmos de ordenación: InsertSort

Ejemplo: l = [4, 7, 3, 2]

Analizo 4: **4** 7 3 2 → ok!
Analizo 7: 4 **7** 3 2 → ok!

Analizo 3: 4 7 **3** 2 → 7 > 3 intercambio!
Analizo 3: 4 **3** 7 2 → 4 > 3 intercambio!
Analizo 3: **3** 4 7 2 → 4 > 3 ok!

Analizo 2: 3 4 7 **2** → 7 > 2 intercambio!
Analizo 2: 3 4 **2** 7 → 4 > 2 intercambio!
Analizo 2: 3 **2** 4 7 → 3 > 2 intercambio!
Analizo 2: **2** 3 4 7 → ok!

Fin: 2 3 4 7

Algoritmos de ordenación: InsertSort

Algoritmo:

Algoritmos de ordenación: InsertSort

Algoritmo:

```
1  def insertSort(l):
2    # Recorro los elementos de l a partir de 1
3    for i in range(1,len(l)):
4      # Guardo valor y posición actual
5      auxiliar = l[i]
6      j = i
7      # Mientras el elemento de la izquierda sea menor
       lo intercambio por j
8      while( j > 0 and auxiliar < l[j-1]):
9        l[j] = l[j-1]
10       j -= 1
11     l[j] = auxiliar
12
13   # Retornamos lista ordenada
14   return l
```

Algoritmos de ordenación: InsertSort

¿Cuál es la complejidad de InsertSort?

Algoritmos de ordenación: InsertSort

¿Cuál es la complejidad de InsertSort?

Sea $n = len(l)$ y k una constante, entonces el número de instrucciones aproximado (en el peor caso) es:

$$k((n-1) + (n-2) + ... + 1)$$

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

Algoritmos de ordenación en Python

Python permite ordenar listas mediante el método **sort()**.

```
1 l = [5,2,3,7,6,4,2]
2 l.sort()
3 print(l)  # >>> [2, 2, 3, 4, 5, 6, 7]
```

Algoritmos de ordenación en Python

Python permite ordenar listas mediante el método **sort()**.

```
1 l = [5,2,3,7,6,4,2]
2 l.sort()
3 print(l)  # >>> [2, 2, 3, 4, 5, 6, 7]
```

... y es muy eficiente $O(n \log(n))$.

Algoritmos de ordenación en Python

¿Por qué vimos estos algoritmos?

Algoritmos de ordenación en Python

¿Por qué vimos estos algoritmos?

- Para hablar de complejidad computacional.
- Son un buen ejercicio práctico.

Algoritmos de ordenación en Python

¿Por qué vimos estos algoritmos?

- Para hablar de complejidad computacional.
- Son un buen ejercicio práctico.
- Nos gusta preguntarlos en pruebas.

Algoritmos de ordenación en Python

¿Por qué vimos estos algoritmos?

- Para hablar de complejidad computacional.
- Son un buen ejercicio práctico.
- Nos gusta preguntarlos en pruebas.
- Existen casos donde **sort()** no es suficiente.

Algoritmos de ordenación en Python

Ordene una lista de usuarios (tuplas: nombre, apellido, edad) según su apellido.

Algoritmos de ordenación en Python

Ordene una lista de usuarios (tuplas: nombre, apellido, edad) según su apellido.

```
1 from random import randrange
2
3 u1 = ("Sofía","Álvarez", randrange(18,22))
4 u2 = ("Manuel", "Aylwin", randrange(18,22))
5 u3 = ("Nicolás", "Balbontín", randrange(18,22))
6 u4 = ("Thomas", "Escalona", randrange(18,22))
7 u5 = ("Juan Pablo", "Bravo", randrange(18,22))
8
9 l = [u1,u2,u3,u4,u5]
```

Algoritmos de ordenación en Python

Ordene una lista de usuarios (tuplas: nombre, apellido, edad) según su apellido.

```
1 from random import randrange
2
3 u1 = ("Sofía","Álvarez", randrange(18,22))
4 u2 = ("Manuel", "Aylwin", randrange(18,22))
5 u3 = ("Nicolás", "Balbontín", randrange(18,22))
6 u4 = ("Thomas", "Escalona", randrange(18,22))
7 u5 = ("Juan Pablo", "Bravo", randrange(18,22))
8
9 l = [u1,u2,u3,u4,u5]
```

¿Cómo ordeno l?

Algoritmos de ordenación en Python

Con SelectSort().

```
1  def selectSort(l):
2    for i in range(len(l)-1):
3      pos = i
4      for j in range(pos+1,len(l)):
5        if(l[pos][1] > l[j][1]):
6          pos = j
7      auxiliar = l[pos]
8      l[pos] = l[i]
9      l[i] = auxiliar
10
11   return l
```

Algoritmos de ordenación en Python

Con InsertSort().

```
1  def insertSort(l):
2    for i in range(1,len(l)):
3      auxiliar = l[i]
4      j = i
5      while( j > 0 and auxiliar[1] < l[j-1][1]):
6        l[j] = l[j-1]
7        j -= 1
8      l[j] = auxiliar
9
10   return l
```

Ejercicios

1) Modifique el código de SelectSort() para que ordene la lista en forma no decreciente.

2) Modifique el código de InsertSort() para que ordene la lista en forma no decreciente.

3) Ordene la lista de usuario por edad.

4) Ordene la lista de usuario por apellido, y rompa empates mediante el nombre.