Análisis y Diseño de Algoritmos.

Sesión 2. 22 de agosto de 2017.

Maestría en Sistemas Computacionales.

Algoritmos de ordenamiento iterativos

- ¿Para qué los estudiaremos?
  - Para ejercitar el diseño y análisis *a priori* y *a posteriori* de algoritmos iterativos que implican casos: mejor, peor y promedio.
  - Se asemejan a otros algoritmos que puedan ser de mayor interés.
- ¿Cuáles estudiaremos?
  - Tres de complejidad cuadrática
  - Tres de complejidad quasi-lineal
  - Uno de complejidad lineal

Algoritmos de ordenamiento de complejidad cuadrática

- ¿Para qué los estudiaremos?
  - Entender la terminología y mecanismos básicos de los algoritmos de ordenamiento.
  - Se pueden extender a mejor métodos de propósito general.
- ¿Cuándo conviene usarlos? (se cumple al menos uno)
  - Cuando se ordenan pocos datos.
  - Cuando los datos están casi ordenados.
  - Cuando hay muchos datos repetidos.

Algoritmos de ordenamiento de complejidad cuadrática

- ¿Cómo los identificamos?
  - El número de pasos que necesitan para ordenar N elementos aleatorios es proporcional a N^2.
    - Comparan cada elemento contra todos o casi todos los demás.
- ¿Qué ventajas tienen sobre los algoritmos más rápidos?
  - Facilidad en la implementación.
  - La mayoría son **estables**.
    - Si una lista de alumnos está ordenada alfabéticamente y se ordena por promedio: los que tienen mismo promedio siguen en orden alfabético.
  - No necesitan memoria extra en el orden de N.

Algoritmos de ordenamiento de complejidad cuadrática

- ¿Cuál método de ordenamiento cuadrático será mejor?
  1. Selección
  2. Inserción
  3. Burbuja
- Definamos criterios de eficiencia cuantitativos que sean independiente de la máquina:
  - Número de comparaciones entre los datos
  - Número de movimientos de los datos (lectura o escritura)

Métodos útiles

- Por simplicidad, ordenaremos arreglos de enteros.
- Para comprobar la precisión de los algoritmos a implementar, programemos primero los siguientes métodos:
  - `static int[] createIntArray(int N, int min, int max)`
    - *Crea un arreglo de N enteros. Cada entero es un aleatorio en [min...max]*
  - `static void printArray(int[] array)`
    - *Imprime en consola el contenido del arreglo: 5, 75, -12, 0, 8*
  - `static void swap(int[] array, int index1, int index2)`
    - *Intercambia los elementos en las posiciones* index1, index2 *de un arreglo dado.*
  - `static boolean isSorted(int[] array)`
    - *Devuelve* verdadero *si el arreglo está ordenado;* falso, *en otro caso.*
    - *¿Cuál es la complejidad temporal de este algoritmo?*

Ordenamiento por Selección

http://cs.armstrong.edu/liang/animation/web/SelectionSortNew.html

- Encuentra el elemento más pequeño del arreglo y lo intercambia por el que está en la primer posición.
- Encuentra el segundo elemento más pequeño (o el más pequeño de los N – 1 restantes) y lo intercambia por el segundo elemento, y así sucesivamente.

```
M U R C I E L A G O
A U R C I E L M G O

A U R C I E L M G O
A C R U I E L M G O

A C R U I E L M G O
A C E U I R L M G O …
```

Análisis del algoritmo Selección

- ¿Cuántas **comparaciones** efectúa el algoritmo de Selección si el arreglo está desordenado?

```
static void selectionSort(int[] array)
   for(int i = 0; i < array.length - 1; i ++) {
       int min_index = i;
       for(int j = i + 1; j < array.length; j ++)
           if(array[j] < array[min_index]) min_index = j;
       if(i != min_index) swap(array, i, min_index);
   }
}
```

Análisis del algoritmo
Selección

- El ciclo externo se ejecuta N – 1 veces. El ciclo interno N – i veces.
  - En la iteración 1 del ciclo externo se efectúan N – 1 comparaciones.
  - En la iteración 2 del ciclo externo se efectúan N – 2 comparaciones.
  - En la iteración (N – 1) del ciclo externo se efectúa 1 comparación.

```
for(int i = 0; i < array.length - 1; i ++) {
    int min_index = i;
    for(int j = i + 1; j < array.length; j ++)
        if(array[j] < array[min_index]) min_index = j;
    if(i != min_index) swap(array, i, min_index);
}
```

Análisis del algoritmo
Selección

- En total se efectúan: $1 + 2 + \ldots (N - 2) + (N - 1)$ comparaciones.
- Recordemos que: $1 + 2 + \ldots\ N = \frac{1}{2} N(N+1)$
- $\therefore$ Comparaciones $= \frac{1}{2}(N - 1)(N) = 0.5N^2 - 0.5N \in O(N^2)$
- ¿Cuál es la complejidad espacial?
- ¿Y si el arreglo estuviera ordenado?
- ¿Y si el arreglo estuviera invertido?

Análisis del algoritmo Selección

- ¿Cuántos **movimientos** efectúa el algoritmo de Selección si el arreglo está ordenado, invertido y desordenado?

```
static void selectionSort(int[] array)
   for(int i = 0; i < array.length - 1; i ++) {
      int min_index = i;
      for(int j = i + 1; j < array.length; j ++)
         if(array[j] < array[min_index]) min_index = j;
      if(i != min_index) swap(array, i, min_index);
   }
}
```

Análisis del algoritmo
Selección

- Si el arreglo está **ordenado**, el índice del elemento más pequeño, min_index, siempre será igual al índice del ciclo externo i.
- Nunca ejecutará el intercambio.
- ∴ el número de movimientos es: $0 \in O(K)$

```
static void selectionSort(int[] array)
   for(int i = 0; i < array.length - 1; i ++) {
       int min_index = i;
       for(int j = i + 1; j < array.length; j ++)
           if(array[j] < array[min_index]) min_index = j;
       if(i != min_index) swap(array, i, min_index);
   }
}
```

Análisis del algoritmo
Selección

- Si el arreglo está **invertido**, en la primera mitad del ciclo externo, min_index siempre será diferente a i: un intercambio por iteración.
- Al comenzar la segunda mitad, el arreglo ya estará ordenado.
- Recordemos que el método `swap` realiza tres movimientos.
- $\therefore$ el número de movimientos es: $3(\frac{1}{2}N) = 1.5N \in O(N)$.
- ¿Y si el arreglo está **desordenado**?
  - Esperaríamos 0.75N movimientos en promedio.
  - Para este caso, se sugiere un análisis **a posteriori**.

Ordenamiento por Inserción

http://cs.armstrong.edu/liang/animation/web/InsertionSortNew.html

- Comienza con el segundo elemento.
- Recorre un elemento a la izquierda tantas posiciones hasta encontrar un elemento más pequeño que él
- Los elementos del lado izquierdo ya están ordenados.

```
M U R C I E L A G O
M U U C I E L A G O
M R U C I E L A G O

M R U C I E L A G O
M R U U I E L A G O
M R R U I E L A G O
M M R U I E L A G O
C M R U I E L A G O …
```

Ordenamiento por Burbuja

http://cs.armstrong.edu/liang/animation/web/BubbleSortNew.html

- Recorre la lista de izquierda a derecha intercambiando elementos adyacentes, si es necesario.
  - El elemento más grande quedará colocado al final.
- Se repite el recorrido sin comparar el último elemento del anterior.
  - Si en una pasada no hubo intercambios, el arreglo está ordenado ☺.

```
M U R C I E L A G O
M R U C I E L A G O
M R C U I E L A G O
M R C I U E L A G O
…
M R C I E L A G O U …
```

Tarea (parte 1)

- Realizar un análisis *a priori* para determinar, en función de N, cuántas comparaciones y movimientos se efectúan para ordenar arreglos ordenados e invertidos para los algoritmos **Inserción** y **Burbuja**.
- Para arreglos aleatorios, realizar un análisis *a posteriori* por cada uno de los tres algoritmos vistos:
  a) Para N = 1 a 200 elementos, realice 100N invocaciones al algoritmo con diferentes arreglos aleatorios.
  b) Calcular el promedio de comparaciones y movimientos por cada N.
  c) Genere dos gráficas de dispersión (X, Y1), (X, Y2) y obtenga la ecuación de la línea de tendencia. X = N, Y1 = promedio de comparaciones, Y2 = promedio de movimientos

Tarea (parte 1)

- Incluir en el documento: análisis *a priori y posteriori en el caso aleatorio* y las seis gráficas .
- Además, llenar la siguiente tabla que resume los hallazgos:

Algoritmo	Criterio	Ordenado (a priori)	Aleatorio (posteriori)	Invertido (a priori)
Selección	Comparaciones	$0.5N^2 - 0.5N \in O(N^2)$		$0.5N^2 - 0.5N \in O(N^2)$
	Movimientos	$0 \in O(K)$		$1.5N \in O(N)$
Inserción	Comparaciones			
	Movimientos			
Burbuja	Comparaciones			
	Movimientos			

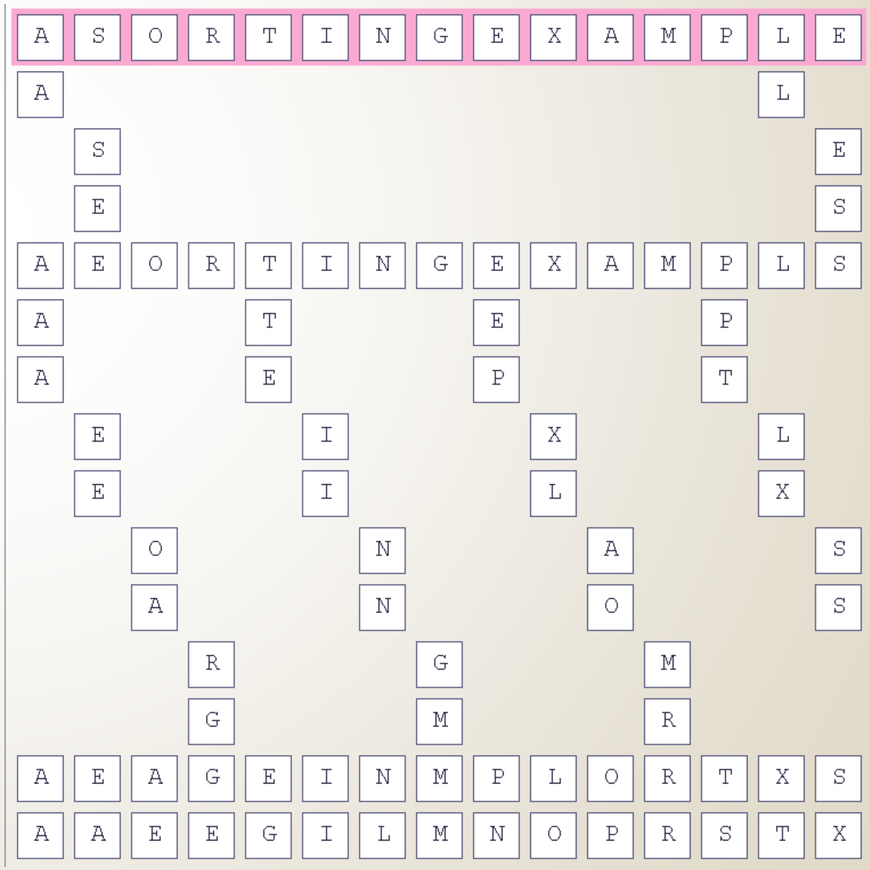
Ordenamiento por Shell

- El método por **Inserción** es lento porque intercambia sólo elementos adyacentes:
  - Si el elemento más pequeño está hasta el final, se necesitan N pasos para ponerlo en su lugar.
- **Shell** es una extensión a **Inserción** que acelera el proceso al intercambiar elementos que están lejos entre sí.
- El sello de este algoritmo es:
  - En una pasada, los elementos k, k+h, k+2h, k+3h,… deben formar un arreglo ordenado, para todo $0 \leq k < h$
  - En cada pasada se disminuye el valor de h hasta llegar a 1
  - ¿Cuánto vale h al inicio? ¿Cómo hay que disminuirlo?

Ordenamiento por Shell

2 / 2

- Algunos autores sugieren la secuencia:
  - 1093, 364, 121, 40, **13, 4, 1**
  - La relación de derecha a izquierda es 3k + 1.
  - Se alternan pares y nones.
  - El tiempo de ejecución se reduce a menos de 1% con respecto al método **Inserción.**
  - Una secuencia mala es:
    64, 32, 16, 8, 4, 2, 1
  - No compara pares con nones.



Tarea (parte 2)

- [PILON] (20 puntos) Mediante un <u>análisis *a posteriori*</u>, demostrar que la <u>complejidad temporal de **Shell**</u> en su caso promedio está en $O(N^{1.26})$.

  (Metodología sugerida)

  - Utilice arreglos de N = 1,000 hasta 60,000 en incrementos de 1,000
  - Registre el promedio de comparaciones de ${}^{N}/_{100}$ corridas por cada N.
  - Obtenga la ecuación a partir de una gráfica de dispersión.
  - En el reporte pegar el código identado a colores y con posibilidad de copiarse.