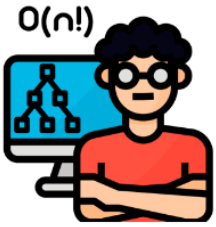
Instituto Politécnico Nacional

Escuela Superior de Cómputo

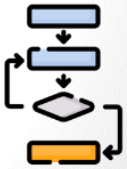


Análisis de Algoritmos

Practica 01: Pruebas a posteriori (Algoritmos de Ordenamiento)

Prof. Edgardo Adrián Franco Martínez
http://eafranco.com
edfrancom@ipn.mx
@edfrancom edfrancom

https://creativecommons.org/licenses/by-nc-sa/4.0/deed.es



Objetivo

Realizar un análisis de algoritmos a posteriori de los algoritmos de **ordenamiento** más **conocidos** en la computación y realizar una aproximación a sus funciones de complejidad temporal.

Definición del problema

- Con base en el archivo de entrada proporcionado que tiene hasta **10,000,000 de números diferentes**; ordenar bajo **los siguientes 10 métodos de ordenamiento** y **comparar experimentalmente las complejidades** aproximadas según se indica en las actividades a reportar.
  - Burbuja *(Bubble Sort)*
    - Burbuja Simple
    - Burbuja Optimizada 1
    - Burbuja Optimizada 2
  - Inserción *(Insertion Sort)*
  - Selección *(Selection Sort)*
  - Shell *(Shell Sort)*
  - Ordenamiento con árbol binario de búsqueda *(Tree Sort)*
  - Ordenamiento por mezcla (*Merge Sort*)
  - Ordenamiento rápido (*Quick Sort*)
  - Ordenamiento por montículos (*Heap Sort*)

2 4 6 1 3 5 7

Ordenamiento Burbuja

1 3 2 4 5 6 7 8

Burbuja Simple

- Funciona revisando cada elemento de la lista que va a ser ordenada con el siguiente, intercambiándolos de posición si están en el orden equivocado. Es necesario revisar varias veces toda la lista hasta que no se necesiten más intercambios, lo cual significa que la lista está ordenada.

- El método de la burbuja es uno de los mas simples, es tan fácil como comparar todos los elementos de una lista contra todos, si se cumple que uno es mayor o menor a otro, entonces los intercambia de posición.

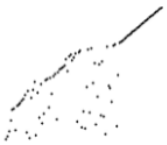
- Este algoritmo obtiene su nombre de la forma con la que suben por la lista los elementos durante los intercambios, como si fueran pequeñas "burbujas". También es conocido como el método del intercambio directo.

Burbuja Simple

- El método de burbuja más simple sin pensar como optimizar simplemente es comparar pares de números iterando el número de elementos que tenemos.

```
Algoritmo BurbujaSimple(A,n)
       para i=0 hasta n-2 hacer
              para j=0 hasta n-2 hacer
                     si (A[j]>A[j+1]) entonces
                            aux = A[j]
                            A[j] = A[j+1]
                            A[j+1] = aux
                     fin si
              fin para
       fin para
fin Algoritmo
```

El arreglo A indexa desde 0 hasta n-1 --- A[0,1,2,…,n-1]



5 6 3 1 8 7 2 4

Burbuja Optimizada 1

- Como al final de cada iteración el elemento mayor queda situado en su posición, ya no es necesario volverlo a comparar con ningún otro número, reduciendo así el número de comparaciones por iteración.

```
Algoritmo BurbujaSimple(A,n)
       para i=0 hasta n-2 hacer
              para j=0 hasta (n-2)-i hacer
                     si (A[j]>A[j+1]) entonces
                            aux = A[j]
                            A[j] = A[j+1]
                            A[j+1] = aux
                     fin si
              fin para
       fin para
fin Algoritmo
```

El arreglo A indexa desde 0 hasta n-1 --- A[0,1,2,…,n-1]

Burbuja Optimizada 2

- Pude existir la posibilidad de realizar iteraciones de más si el arreglo ya fue ordenado totalmente, por lo que es posible llevar una centinela para saber si hubo cambios o no en una iteración.

```
Algoritmo BurbujaOptimizada(A,n)
        cambios = SI
        i=0
        mientras i<= n-2 && cambios != NO hacer
                cambios = NO
                para j=0 hasta (n-2)-i hacer
                        si(A[j] < A[j+1]) entonces
                                aux = A[j]
                                A[j] = A[j+1]
                                A[j+1] = aux
                                cambios = SI
                        fin si
                fin para
                 i= i+1
        fin mientras
fin Algoritmo
```

El arreglo A indexa desde 0 hasta n-1 -- A[0,1,…, n-1]

Ordenamiento por inserción

6 5 3 1 8 7 2 4

- Es una manera muy natural de ordenar para un ser humano, y puede usarse fácilmente para ordenar un mazo de cartas numeradas en forma arbitraria.

- Inicialmente se tiene un solo elemento, que obviamente es un conjunto ordenado. Después, cuando hay *k* elementos ordenados de menor a mayor, se toma el elemento *k+1* y se compara con todos los elementos ya ordenados, deteniéndose cuando se encuentra un elemento menor (todos los elementos mayores han sido desplazados una posición a la derecha) o cuando ya no se encuentran elementos (todos los elementos fueron desplazados y este es el más pequeño). En este punto se ***inserta*** el elemento ***k+1*** debiendo desplazarse los demás elementos.

```
Algoritmo Insercion(A,n)
{
        para i=0 hasta n-1 hacer
                j=i
                temp=A[i]
                mientras(j>0) && (temp<A[j-1]) hacer
                        A[j]=A[j-1]
                        j--
                fin mientras
                A[j]=temp
        fin para
fin Algoritmo
```

3	5	6	1	8	7	2	4

El arreglo A indexa desde 0 hasta n-1 -- A[0,1,…, n-1]

Ordenamiento por selección

0
5
2
6
9
3
1
4
8
7

- Se basa en buscar el mínimo elemento de la lista e intercambiarlo con el primero, después busca el siguiente mínimo en el resto de la lista y lo intercambia con el segundo, y así sucesivamente.

- **Algoritmo**
  - Buscar el mínimo elemento entre una posición i y el final de la lista Intercambiar el mínimo con el elemento de la posición *i*.

```
Algoritmo Seleccion(A,n)
       para k=0 hasta n-2 hacer
              p=k
              para i=k+1 hasta n-1 hacer
                     si A[i]<A[p] entonces
                            p=i
                     fin si
              fin para
              temp = A[p]
              A[p] = A[k]
              A[k] = temp
       fin para
fin Algoritmo
```

El arreglo A indexa desde 0 hasta n-1 -- A[0,1,…, n-1]

0	5
1	2
2	6
3	9
4	3
5	1
6	4
7	8

Ordenamiento Shell

- El Shell es una generalización del ordenamiento por inserción, teniendo en cuenta dos observaciones:
  1. El ordenamiento por inserción es eficiente si la entrada está "casi ordenada".
  2. El ordenamiento por inserción es ineficiente, en general, porque mueve los valores sólo una posición cada vez.

- El algoritmo Shell mejora el ordenamiento por inserción comparando elementos separados por un espacio de varias posiciones. Esto permite que un elemento haga "pasos más grandes" hacia su posición esperada. Los pasos múltiples sobre los datos se hacen con tamaños de espacio cada vez más pequeños. El último paso del ordenamiento Shell es un simple ordenamiento por inserción, pero para entonces, ya está garantizado que los datos del vector están casi ordenados.

- Shell propone que se haga sobre el arreglo una serie de ordenaciones basadas en la inserción directa, pero dividiendo el arreglo original en varios sub-arreglos tales que cada elemento esté separado k elementos del anterior (a esta separación a menudo se le llama ***salto*** o ***gap***)

- Se debe empezar con k=n/2, siendo n el número de elementos del arreglo, y utilizando siempre la división entera (**TRUNC**)

- Después iremos variando k haciéndolo más pequeño mediante sucesivas divisiones por 2, hasta llegar a k=1.


```
Algoritmo Shell(A,n)
        k = TRUNC(n/2)
        mientras k >=1 hacer
                b=1
                mientras b!=0 hacer
                        b=0
                        para i=k hasta n-1 hacer
                                si A[i-k]>A[i] entonces
                                        temp=A[i]
                                        A[i]=A[i-k]
                                        A[i-k]=temp
                                        b=b+1
                                fin si
                        fin para
                fin mientras
                k=TRUNC(k/2)
        fin mientras
fin Algoritmo
```

El arreglo A indexa desde 0 hasta n-1 -- A[0,1,…, n-1]

Ordenamiento con un Árbol binario de búsqueda (ABB)

- El ordenamiento con la ayuda de un árbol binario de búsqueda es muy simple debido a que solo requiere de dos pasos simples.
  1. Insertar cada uno de los números del vector a ordenar en el árbol binario de búsqueda.
  2. Remplazar el vector en desorden por el vector resultante de un recorrido ***InOrden*** del Árbol Binario de Búsqueda (ABB), el cual entregara los números ordenados.

- La eficiencia de este algoritmo esta dada según la eficiencia en la implementación del árbol binario de búsqueda, lo que puede resultar mejor que otros algoritmos de ordenamiento.

```
Algoritmo OrdenaConABB(A,n)

        para i=0 hasta n-1 hacer
                Insertar(ABB,A[i]);
        fin para

        GuardarRecorridoInOrden(ABB,A);

fin Algoritmo
```

El arreglo A indexa desde 0 hasta n-1 -- A[0,1,…, n-1]

```
Algoritmo MergeSort(A, p, r)
   si p < r entonces
      q = parteEntera((p+r)/2)
      MergeSort(A, p, q)
      MergeSort(A, q+1,r)
      Merge(A, p, q, r)
   fin si
fin Algoritmo
```

```
Algoritmo Merge(A, p, q, r)
   l=r-p+1, i=p, j=q+1
   para k=0 hasta l hacer
      si i<=q y j<=r entonces
         si A[i]<A[j] entonces
            C[k]=A[i]
            i++
         sino entonces
            C[k]=A[j]
            j++;
         fin si
      sino si i<=q entonces
         C[k]=A[i]
         i++
      sino entonces
         C[k]=A[j]
         j++
      fin si
   A[p-r]=C[]
fin Algoritmo
```

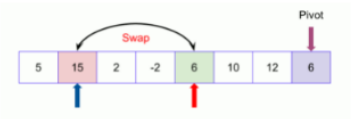
El arreglo A indexa desde 0 hasta n-1 -- A[0,1,…, n-1]

Ordenamiento rápido (*Quick Sort*)

- El algoritmo trabaja de la siguiente forma:

1. Elegir un elemento del conjunto de elementos a ordenar, al que llamaremos pivote.
2. Resituar los demás elementos de la lista a cada lado del pivote, de manera que a un lado queden todos los menores que él, y al otro los mayores. Los elementos iguales al pivote pueden ser colocados tanto a su derecha como a su izquierda, dependiendo de la implementación deseada. En este momento, el pivote ocupa exactamente el lugar que le corresponderá en la lista ordenada.
3. La lista queda separada en dos sub-listas, una formada por los elementos a la izquierda del pivote, y otra por los elementos a su derecha.
4. Repetir este proceso de forma recursiva para cada sub-lista mientras éstas contengan más de un elemento. Una vez terminado este proceso todos los elementos estarán ordenados.

```
Algoritmo QuickSort(A, p, r)
   si p < r entonces
      j = Pivot(A,p,r)
      QuickSort(A, p, j-1)
      QuickSort(A, j+1,r)
   fin si
fin Algoritmo
```




```
Algoritmo Pivot(A, p, r)
   piv=A[p], i=p+1, j=r
   mientras (i<j)
        mientras A[i]<= piv y i<r hacer
            i++
        mientras A[j]> piv hacer
            j--
        Intercambiar(A,i,j)
   fin mientras
   Intercambiar(A,p,j)
   regresar j
fin Algoritmo
```

```
Algoritmo Intercambiar(A, i, j)
  temp= A[j]
  A[j]=A[i]
  A[i]=temp
fin Algoritmo
```

El arreglo A indexa desde 0 hasta n-1 -- A[0,1,…, n-1]

Ordenamiento por montículos (*Heap Sort*)

- Este ordenamiento requiere de una estructura de datos de tipo árbol llamada montículo (*Heap*) y dadas la propiedades de este, el algoritmo de ordenamiento es muy simple debido a que solo requiere de dos pasos simples.
  1. Insertar cada uno de los números del vector a ordenar en el montículo (montículo de menores en caso de ordenar ascendentemente o montículo de mayores en caso de ordenar descendentemente).
  2. Remplazar el vector en desorden por el vector resultante de un ciclo extrayendo un elemento a la vez del montículo.

- La eficiencia de este algoritmo esta dada según la eficiencia en la implementación del montículo, lo que puede resultar mejor que otros algoritmos de ordenamiento.

```
Algoritmo HeapSort(A,n)

        para i=0 hasta n-1 hacer
                Insertar(Heap,A[i]);
        fin para

        para i=0 hasta n-1 hacer
                A[i]=Extraer(Heap);
        fin para

fin Algoritmo
```

El arreglo A indexa desde 0 hasta n-1 -- A[0,1,…, n-1]

Actividades

1. Programar en ANSI C, cada uno de los 10 algoritmos de ordenamiento mencionados.

```
inserción(t)
for i=1 to n
    x= t[i]
    j=i-1
    while j > 0 and x<t[j]
       t[j+1]=t[j]
       j=j-1
    t[j+1]=x
```

2. Adaptar el programa para que sea capaz de recibir un parámetro **"n"** que indica el numero de enteros a ordenar a partir de un archivo con máximo 10,000,000 de números en desorden.

Análisis de Algoritmos
Practica 01: Pruebas a posteriori (Algoritmos de Ordenamiento)
Prof. Edgardo Adrián Franco Martínez

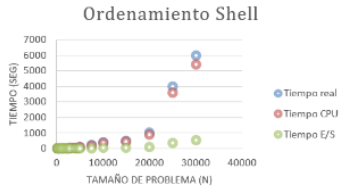
3. Medir el tiempo que tarda cada algoritmo en ordenar 500,000 números **(*n=500,000)*** y compare los tiempos *(de CPU) y (tiempo real)* de cada algoritmo gráficamente (*Comparativa de grafica de barras, una para tiempos de CPU y otras para tiempo real*) y llenar la tabla siguiente.

   - Auxiliarse de la librería de C proporcionada para medir tiempos de ejecución bajo Linux.

Ordenamiento n=500,000

Algoritmo	Tiempo Real	Tiempo CPU	Tiempo E/S	% CPU/Wall

4. Realizar un análisis temporal **para cada algoritmo, ordenando**:
   - Los primeros 100, 1000, 5000, 10000, 50000, 100000, 200000, 400000, 600000, 800000, 1000000, 2000000, 3000000, 4000000, 5000000, 6000000, 7000000, 8000000, 9000000 y 10000000.
   - Graficar el comportamiento temporal de cada algoritmo



**Si no es posible probar el total de n para todos los algoritmos (debido a tiempos de cómputo altos). Colocar al menos 20 puntos de prueba distribuidos en el rango hasta la máxima n alcanzada*

5. Graficar una comparativa de los 10 algoritmos de ordenamiento implementados (únicamente ***Tiempo real***).

6. Realizar una **aproximación a la función del comportamiento temporal** (***tiempo real***), de cada uno de los algoritmos probados con los datos probados según el punto 4. **Utilizar código de Matlab que se proporciona*.*
   - Paso 1: Aproximar cada algoritmo con un polinomio de grado 1, 2, 3 u 6. Mostrar su gráfica de cada aproximación. Llamada a `polyfit()`
   - Paso 2: Si se nota que un polinomio no se ajusta al comportamiento experimental con ninguno de los 4 grados probados entonces buscar una aproximación no polinomial.

Otros modelos que se pueden encontrar al ajustar los puntos t e y

Función	Llamada a `polyfit()`
$f(t,(x_1,x_2)) = x_1 \cdot t^{x_2}$	`p=polyfit(log(t), log(y),1)`
$f(t,(x_1,x_2)) = x_1 \cdot e^{x_2 t}$	`p=polyfit(t, log(y),1)`
$f(t,(x_1,x_2)) = x_1 \cdot ln(t) + x_2$	`p=polyfit(log(t),y,1)`
$f(t,(x_1,x_2)) = \frac{1}{x_1(t+x_2}$	`p=polyfit(t,1./y,1)`

Ajuste de datos con Matlab
https://es.mathworks.com/help/matlab/ref/polyfit.html
http://www.sc.ehu.es/sbweb/fisica3/datos/regresion/regresion_1.html

7. Mostrar gráficamente la **comparativa de las aproximaciones de la función de complejidad temporal** encontradas para cada algoritmo *(todas las aproximaciones para un algoritmo en una gráfica)* y determinar (*justificando*) cuál es la mejor aproximación para cada algoritmo *(10 comparativas)*.

8. **Seleccione una aproximación a la función complejidad** para cada algoritmo, que mejor modela la complejidad y grafíquelos en una comparativa *(1 grafica que compara las 10 funciones determinadas)*

9. Determine con base en las aproximaciones seleccionadas *(según el punto 8)* cual será el tiempo real de cada algoritmo para ordenar 15000000, 20000000, 500000000, 1000000000 y 500000000 números.

10. Finalmente responda a las siguientes preguntas:
    i. ¿Cuál de los 10 algoritmos es más fácil de implementar?
    ii. ¿Cuál de los 10 algoritmos es el más difícil de implementar?
    iii. ¿Cuál algoritmo tiene menor complejidad temporal?
    iv. ¿Cuál algoritmo tiene mayor complejidad temporal?
    v. ¿Cuál algoritmo tiene menor complejidad espacial? ¿Por qué?
    vi. ¿Cuál algoritmo tiene mayor complejidad espacial? ¿Por qué?
    vii. ¿El comportamiento experimental de los algoritmos era el esperado? ¿Por que?
    viii. ¿Existió un entorno controlado para realizar las pruebas experimentales? ¿Cuál fue?
    ix. ¿Facilito las pruebas mediante scripts u otras automatizaciones? ¿Cómo lo hizo?
    x. ¿Qué recomendaciones darían a nuevos equipos para realizar esta practica?

Rubrica de evaluación de la practica

Indicador	Excelente	Muy bien	Bien	Deficiente
Entrega en fecha y forma	Se entrego dentro del plazo y en la sesión dispuesta para ello estando presentes todos los integrantes del equipo, con documentación suficiente.	Se entrego dentro del plazo y en la sesión dispuesta para ello estando presentes todos los integrantes del equipo, con documentación mínima pero aceptable.	Se entrego dentro del plazo y en la sesión dispuesta para ello estando presentes todos los integrantes del equipo, con documentación mínima e incompleta.	Se entrego dentro del plazo y en la sesión dispuesta para ello estando presentes todos los integrantes del equipo, con sin documentación.
Redacción	No hay errores de gramática, ortografía y puntuación y la redacción es coherentemente	No hay errores de gramática, ortografía y puntuación, pero la redacción presenta incoherencias	Pocos errores de gramática, ortografía y puntuación	Muchos errores de gramática, ortografía y puntuación
Cantidad de información	Todos las secciones son tratadas de manera clara y precisa, según lo solicitado.	La mayoría de los secciones son tratados de manera clara y precisa	Hay secciones no incluidas o no diferenciadas.	Hay secciones no incluidas o no diferenciadas y no cumplen lo solicitado.
Calidad de la información	La información está claramente relacionada con el tema principal y proporciona varias ideas secundarias y/o ejemplos	La información da respuestas a las preguntas principales, y solo da algunos detalles y/o ejemplos	La información da respuestas a las preguntas principales, pero no da detalles y/o ejemplos	La información tiene poco o nada que ver con las preguntas planteadas.
Algoritmos	Los algoritmos están implementados de una manera muy clara de interpretarse en el código y meramente necesario.	Los algoritmos están implementados de una manera clara de interpretarse en el código, con algo de código extra no necesario.	Los algoritmos están implementados de una manera poco clara de interpretarse en el código.	Los algoritmos están implementados de una manera poco clara de interpretarse en el código, con mucho código extra no necesario.
Organización	La información está muy bien organizada con párrafos bien redactados y con subtítulos con estilos adecuados	La información está organizada, pero no se distingue en estilos adecuados	La información está organizada, pero los párrafos no están bien redactados	La información proporcionada no parece estar organizada o es copiada de referencias externas de manera literal

Reporte de practica

- Portada
- Planteamiento del problema
- Entorno de pruebas (Características del hardware y software de la maquina donde se corrieron las pruebas y descripción del entorno controlado)
- Actividades y Pruebas (Verificación de la solución, pruebas y resultados de la práctica según lo solicitado *Preguntas a responder)
- Anexo (Códigos fuente *con colores e instrucciones de compilación)
- Bibliografía (En formato IEEE)

Entrega en plataforma

- **En un solo archivo comprimido** (ZIP, RAR, TAR, JAR o GZIP)
  - Reporte (DOC, DOCX o PDF)
  - Códigos fuente (.C, .H, etc.)
    - **Código documentado:** Titulo, descripción, fecha, versión, autor.
    - *(Funciones y Algoritmos: ¿Qué hace?, ¿Cómo lo hace?, ¿Qué recibe?, ¿Qué devuelve?, ¿Causa de errores?).*
    - OBSERVACIONES
    - *NO enviar ejecutables o archivos innecesarios, las instrucciones de compilación van en el anexo del reporte. (Yo compilare los fuente).
    - *NO enviar archivo de números en desorden ni archivo de números ordenados.

Fechas de entrega

- **Entrega de reporte y códigos**
  - En un solo archivo comprimido.

- **Fecha y hora limite de entrega vía plataforma**
  - **Lunes 28 de Marzo de 2021** a las 23:59:59 hrs.