**UNIVERSIDAD TECNOLOGIA NACIONAL**

**FACULTAD REGIONAL TUCUMAN**

****

**Algoritmo y estructura de datos**

**Trabajo Integrador Grupal - Primer Cuatrimestre**

**TEMA:** *Métodos de ordenamiento* **COMISION:** *1K11*

**ALUMNOS:** *Garcia José Elias*

*Giménez Nahuel Leandro*

**ACTIVIDADES PROPUESTAS**

**1)** ¿Qué es el Análisis Algorítmico?

**2)** Definir Orden de un Algoritmo.

**3)** Analizar los siguientes métodos de ordenamiento:

**a.** Intercambio o burbuja mejorada.

**b.** Inserción o método de la baraja

**c.** Selección o método sencillo

**d.** Rápido o QuickSort

**e.** Por Mezcla o MergeSort

**4)** Diferencia entre los diferentes métodos.

**5)** La defensa será oral y realizada por todos los miembros el grupo. Cualquiera puede empezar y continuar otro compañero seleccionado o no por el docente.

**6)** Pueden utilizar diapositiva, videos, o bien cualquier medio que consideren apropiado para mejorar su presentación.

**7)** La presentación del trabajo se hará a través de la plataforma y la defensa de la misma es en forma oral ante profesor de teoría y en los horarios a convenir con él.

**8)** Se deberá detallar en la última hoja, las fuentes consultadas. (Libros, páginas web, docentes, documentos monográficos, etc.)

**RESOLUCION DE ACTIVIDADES**

1. **El análisis de algoritmos** es una parte importante de la Teoría de complejidad computacional más amplia, que provee estimaciones teóricas para los recursos que necesita cualquier algoritmo que resuelva un problema computacional dado. Estas estimaciones resultan ser bastante útiles en la búsqueda de algoritmos eficientes.

A la hora de realizar un análisis teórico de algoritmos es común calcular su complejidad en un sentido asintótico, es decir, para un tamaño de entrada suficientemente grande. La cota superior asintótica, y las notaciones omegas (cota inferior) y theta (caso promedio) se usan con esa finalidad.

Por ejemplo, la búsqueda binaria decimos que se ejecuta en una cantidad de pasos proporcional a un logaritmo, en O(log(n)), coloquialmente "en tiempo logarítmico". Normalmente las estimaciones asintóticas se utilizan porque diferentes implementaciones del mismo algoritmo no tienen por qué tener la misma eficiencia. No obstante, la eficiencia de dos implementaciones "razonables" cualesquiera de un algoritmo dado están relacionadas por una constante multiplicativa llamada constante oculta.

La medida exacta (no asintótica) de la eficiencia a veces puede ser computada, pero para ello suele hacer falta aceptar supuestos acerca de la implementación concreta del algoritmo, llamada modelo de computación. Un modelo de computación puede definirse en términos de un ordenador abstracto, como la Máquina de Turing, y/o postulando que ciertas operaciones se ejecutan en una unidad de tiempo. Por ejemplo, si al conjunto ordenado al que aplicamos una búsqueda binaria tiene 'n' elementos, y podemos garantizar que una única búsqueda binaria puede realizarse en un tiempo unitario, entonces se requieren como mucho log2 N + 1 unidades de tiempo para devolver una respuesta.

Las medidas exactas de eficiencia son útiles para quienes verdaderamente implementan y usan algoritmos, porque tienen más precisión y así les permite saber cuánto tiempo pueden suponer que tomará la ejecución. Para algunas personas, como los desarrolladores de videojuegos, una constante oculta puede significar la diferencia entre éxito y fracaso.}

Las estimaciones de tiempo dependen de cómo definamos un paso. Para que el análisis tenga sentido, debemos garantizar que el tiempo requerido para realizar un paso se encuentre acotado superiormente por una constante. Hay que mantenerse precavido en este terreno; por ejemplo, algunos análisis cuentan con que la suma de dos números se hace en un paso. Este supuesto puede no estar garantizado en ciertos contextos. Si por ejemplo los números involucrados en la computación pueden ser arbitrariamente grandes, dejamos de poder asumir que la adición requiere un tiempo constante (usando papel y lápiz, compara el tiempo que necesitas para sumar dos enteros de 2 dígitos cada uno y el necesario para hacerlo con dos enteros, pero de 1000 dígitos cada uno).

1. **Orden de los algoritmos**

En computación y matemáticas un algoritmo de ordenamiento es un algoritmo que pone elementos de una lista o un vector en una secuencia dada por una relación de orden, es decir, el resultado de salida ha de ser una permutación —o reordenamiento— de la entrada que satisfaga la relación de orden dada. Las relaciones de orden más usadas son el orden numérico y el orden lexicográfico. Ordenamientos eficientes son importantes para optimizar el uso de otros algoritmos (como los de búsqueda y fusión) que requieren listas ordenadas para una ejecución rápida. También es útil para poner datos en forma canónica y para generar resultados legibles por humanos.

Desde los comienzos de la computación, el problema del ordenamiento ha atraído gran cantidad de investigación, tal vez debido a la complejidad de resolverlo eficientemente a pesar de su planteamiento simple y familiar. Por ejemplo, **BubbleSort** fue analizado desde **1956**. Aunque muchos puedan considerarlo un problema resuelto, nuevos y útiles algoritmos de ordenamiento se siguen inventado hasta el día de hoy (por ejemplo, el ordenamiento de biblioteca se publicó por primera vez en el 2004). Los algoritmos de ordenamiento son comunes en las clases introductorias a la computación, donde la abundancia de algoritmos para el problema proporciona una gentil introducción a la variedad de conceptos núcleo de los algoritmos, como notación de O mayúscula, algoritmos divide y vencerás, estructuras de datos, análisis de los casos peor, mejor, y promedio, y límites inferiores.

**Clasificación:**

Los algoritmos de ordenamiento se pueden clasificar en las siguientes maneras:

La más común es clasificar según el lugar donde se realice la ordenación.

**Algoritmo de ordenamiento interno:** son aquellos que son manejados usando la memoria primaria, es decir la memoria de trabajo o memoria RAM.

**Algoritmo de ordenamiento externo:** Es un término genérico para los algoritmos de ordenamiento que pueden manejar grandes cantidades de información. El ordenamiento externo se requiere cuando la información que se tiene que ordenar no cabe en la memoria principal de una computadora (típicamente la RAM) y un tipo de memoria más lenta (típicamente un disco duro) tiene que utilizarse en el proceso.

Por el tiempo que tardan en realizar la ordenación, dadas entradas ya ordenadas o inversamente ordenadas:

**Algoritmos de ordenación natural:** Tarda lo mínimo posible cuando la entrada está ordenada.

**Algoritmos de ordenación no natural:** Tarda lo mínimo posible cuando la entrada está inversamente ordenada.

**3)a. Método de ordenamiento de intercambio o burbuja mejorada:**

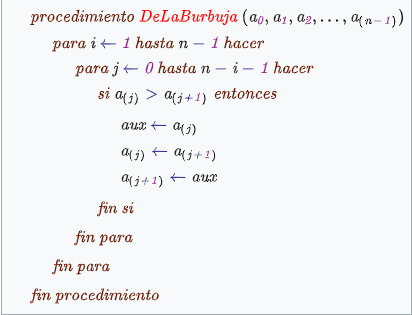
La Ordenación de burbuja (Bubble Sort en inglés) es un sencillo algoritmo de ordenamiento. Funciona revisando cada elemento de la lista que va a ser ordenada con el siguiente, intercambiándolos de posición si están en el orden equivocado.

Es necesario revisar varias veces toda la lista hasta que no se necesiten más intercambios, lo cual significa que la lista está ordenada. 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. Dado que solo usa comparaciones para operar elementos, se lo considera un algoritmo de comparación, siendo uno de los más sencillos de implementar.



(Proceso de ordenamiento de la burbuja mejorada o intercambio)

Una manera simple de expresar el ordenamiento de burbuja en pseudocódigo es la siguiente:

Este algoritmo realiza el ordenamiento o reordenamiento de una lista a de n valores, en este caso de n términos numerados del 0 al n-1; consta de dos bucles anidados, uno con el índice i, que da un tamaño menor al recorrido de la burbuja en sentido inverso de 2 a n, y un segundo bucle con el índice j, con un recorrido desde 0 hasta n-i, para cada iteración del primer bucle, que indica el lugar de la burbuja.

La burbuja son dos términos de la lista seguidos, j y j+1, que se comparan: si el primero es mayor que el segundo sus valores se intercambian.

Esta comparación se repite en el centro de los dos bucles, dando lugar a una lista ordenada. Puede verse que el número de repeticiones solo depende de n y no del orden de los términos, esto es, si pasamos al algoritmo una lista ya ordenada, realizará todas las comparaciones exactamente igual que para una lista no ordenada. Esta es una característica de este algoritmo. Luego veremos una variante que evita este inconveniente.

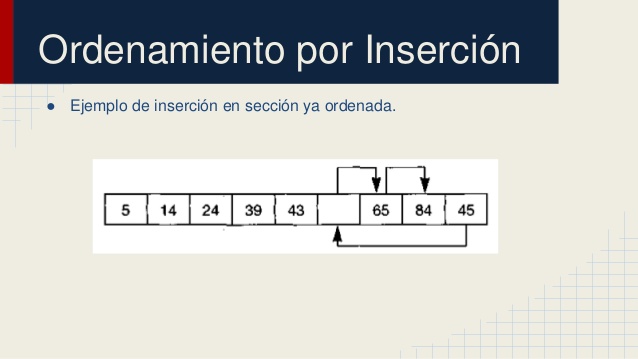
Un ejemplo de método de burbuja en codificación lenguaje C++:



**3)b. Método de ordenamiento de inserción o baraja:**

El ordenamiento por inserción (insertion sort en inglés) 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. Requiere **O(n2)** operaciones para ordenar una lista de **n** elementos.

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.



3)c. **Método de ordenamiento de selección o método sencillo**

El ordenamiento por selección es un algoritmo de ordenamiento que requiere **O(n2)** operaciones para ordenar una lista de n elementos.

Su funcionamiento es el siguiente:

\*Buscar el mínimo elemento de la lista

\*Intercambiarlo con el primero

\*Buscar el siguiente mínimo en el resto de la lista

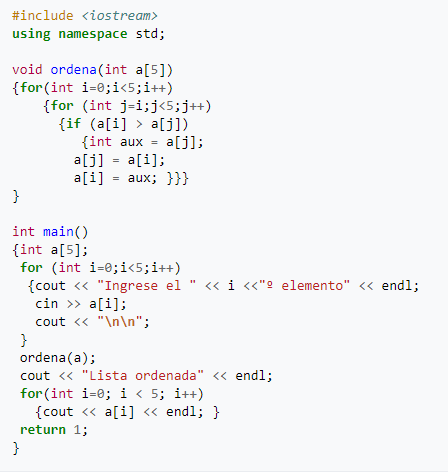
\*Intercambiarlo con el segundo

Y en general:

Buscar el mínimo elemento entre una posición i y el final de la lista

Intercambiar el mínimo con el elemento de la posición i

Un ejemplo de este método programado en un lenguaje C++



3)d. **Método de ordenamiento rápido o quicksort**

El ordenamiento rápido (quicksort en inglés) es un algoritmo de ordenación creado por el científico británico en computación C. A. R. Hoare.

El algoritmo trabaja de la siguiente forma:

**\***Elegir un elemento del conjunto de elementos a ordenar, al que llamaremos pivote.

**\***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.

**\***La lista queda separada en dos sublistas, una formada por los elementos a la izquierda del pivote, y otra por los elementos a su derecha.

**\***Repetir este proceso de forma recursiva para cada sublistas mientras éstas contengan más de un elemento. Una vez terminado este proceso todos los elementos estarán ordenados.

Como se puede suponer, la eficiencia del algoritmo depende de la posición en la que termine el pivote elegido.

**\***En el mejor caso, el pivote termina en el centro de la lista, dividiéndola en dos sublistas de igual tamaño. En este caso, el orden de complejidad del algoritmo es \Omega (n·log n).

**\***En el peor caso, el pivote termina en un extremo de la lista. El orden de complejidad del algoritmo es entonces de O(n²). El peor caso dependerá de la implementación del algoritmo, aunque habitualmente ocurre en listas que se encuentran ordenadas, o casi ordenadas. Pero principalmente depende del pivote, si por ejemplo el algoritmo implementado toma como pivote siempre el primer elemento del array, y el array que le pasamos está ordenado, siempre va a generar a su izquierda un array vacío, lo que es ineficiente.

**\***En el caso promedio, el orden es O(n·log n).

No es extraño, pues, que la mayoría de optimizaciones que se aplican al algoritmo se centren en la elección del pivote

**Demostración de un caso particular**

Supongamos que el número de elementos a ordenar es una potencia de dos, es decir, **n=2k** para algún natural **k**. Inmediatamente **k=log2(n),** donde k es el número de divisiones que realizará el algoritmo.

En la primera fase del algoritmo habrá n comparaciones. En la segunda fase el algoritmo instanciará dos sublistas de tamaño aproximadamente **n/2**. El número total de comparaciones de estas dos sublistas es: **2(n/2) = n**. En la tercera fase el algoritmo procesará 4 sublistas más, por tanto, el número total de comparaciones en esta fase es **4(n/4) = n**.

En conclusión, el número total de comparaciones que hace el algoritmo es:

**N+n+n+n+…+n =kn** donde **k=log2(n)**, por tanto, el Orden de Complejidad del algoritmo en el mejor de los casos es **O(n.log2n)**.

**Técnicas de elección del pivote**

El algoritmo básico del método Quicksort consiste en tomar cualquier elemento de la lista al cual denominaremos como pivote, dependiendo de la partición en que se elija, el algoritmo será más o menos eficiente.

Tomar un elemento cualquiera como pivote tiene la ventaja de no requerir ningún cálculo adicional, lo cual lo hace bastante rápido. Sin embargo, esta elección «a ciegas» siempre provoca que el algoritmo tenga un orden de **O(n²)** para ciertas permutaciones de los elementos en la lista.

Otra opción puede ser recorrer la lista para saber de antemano qué elemento ocupará la posición central de la lista, para elegirlo como pivote. Esto puede hacerse en **O(n)** y asegura que hasta en el peor de los casos, el algoritmo sea **O(n·log n).** No obstante, el cálculo adicional rebaja bastante la eficiencia del algoritmo en el caso promedio.

La opción a medio camino es tomar tres elementos de la lista - por ejemplo, el primero, el segundo, y el último - y compararlos, eligiendo el valor del medio como pivote.

**Técnicas de reposicionamiento**

Una idea preliminar para ubicar el pivote, en su posición final sería contar la cantidad de elementos menores que él, y colocarlo un lugar más arriba, moviendo luego todos esos elementos menores que él a su izquierda, para que pueda aplicarse la recursividad.

Existe, no obstante, un procedimiento mucho más efectivo. Se utilizan dos índices: i, al que llamaremos índice izquierdo, y j, al que llamaremos índice derecho. El algoritmo es el siguiente:

Recorrer la lista simultáneamente con i y j: por la izquierda con i (desde el primer elemento), y por la derecha con j (desde el último elemento).

Cuando lista[i] sea mayor que el pivote y lista[j] sea menor, se intercambian los elementos en esas posiciones.

Repetir esto hasta que se crucen los índices.

El punto en que se cruzan los índices es la posición adecuada para colocar el pivote, porque sabemos que a un lado los elementos son todos menores y al otro son todos mayores (o habrían sido intercambiados).

**Transición a otro algoritmo**

Como se mencionó anteriormente, el algoritmo quicksort ofrece un orden de ejecución O(n²) para ciertas permutaciones "críticas" de los elementos de la lista, que siempre surgen cuando se elige el pivote «a ciegas». La permutación concreta depende del pivote elegido, pero suele corresponder a secuencias ordenadas. Se tiene que la probabilidad de encontrarse con una de estas secuencias es inversamente proporcional a su tamaño.

Los últimos pases de quicksort son numerosos y ordenan cantidades pequeña de elementos. Un porcentaje medianamente alto de ellos estarán dispuestos de una manera similar al peor caso del algoritmo, volviendo a éste ineficiente. Una solución a este problema consiste en ordenar las secuencias pequeñas usando otro algoritmo. Habitualmente se aplica el algoritmo de inserción para secuencias de tamaño menores de 8-15 elementos.

Pese a que en secuencias largas de elementos la probabilidad de hallarse con una configuración de elementos "crítica" es muy baja, esto no evita que sigan apareciendo (a veces, de manera intencionada). El algoritmo introsort es una extensión del algoritmo quicksort que resuelve este problema utilizando heapsort en vez de quicksort cuando el número de recursiones excede al esperado.

Nota: Los tres parámetros de la llamada inicial a Quicksort serán array [0], 0, numero elementos -1, es decir, si es un array de 6 elementos array, 0, 5

**Ejemplo:**

En el siguiente ejemplo se marcan el pivote y los índices i y j con las letras p, i y j respectivamente.

Comenzamos con la lista completa. El elemento pivote será el **4**

5 - 3 - 7 - 6 - 2 - 1 - 4

p

Comparamos con el 5 por la izquierda y el 1 por la derecha.

5 - 3 - 7 - 6 - 2 - 1 - 4

I j p

5 es mayor que 4 y 1 es menor. Intercambiamos:

1 - 3 - 7 - 6 - 2 - 5 - 4

i j p

Avanzamos por la izquierda y la derecha:

1 - 3 - 7 - 6 - 2 - 5 - 4

i j p

3 es menor que 4: avanzamos por la izquierda. 2 es menor que 4: nos mantenemos ahí.

1 - 3 - 7 - 6 - 2 - 5 - 4

i j p

7 es mayor que 4 y 2 es menor: intercambiamos.

1 - 3 - 2 - 6 - 7 - 5 - 4

i j p

Avanzamos por ambos lados:

1 - 3 - 2 - 6 - 7 - 5 - 4

Iyj p

En este momento termina el ciclo principal, porque los índices se cruzaron. Ahora intercambiamos lista[i] con lista[p] (pasos 16-18):

1 - 3 - 2 - 4 - 7 - 5 - 6

p

Aplicamos recursivamente a la sublistas de la izquierda (índices 0 - 2). Tenemos lo siguiente:

1 - 3 - 2

1 es menor que 2: avanzamos por la izquierda. 3 es mayor: avanzamos por la derecha. Como se intercambiaron los índices termina el ciclo. Se intercambia lista[i] con lista[p]:

1 - 2 - 3

El mismo procedimiento se aplicará a la otra sublistas. Al finalizar y unir todas las sublistas queda la lista inicial ordenada en forma ascendente.

1 - 2 - 3 - 4 - 5 - 6 - 7

3)e. Método de ordenamiento por mezcla o MergeSort

El algoritmo de ordenamiento por mezcla (merge sort en inglés) es un algoritmo de ordenamiento externo estable basado en la técnica divide y vencerás.

Fue desarrollado en 1945 por John Von Neumann.1​

Conceptualmente, el ordenamiento por mezcla funciona de la siguiente manera:

* Si la longitud de la lista es 0 ó 1, entonces ya está ordenada. En otro caso:
* Dividir la lista desordenada en dos sublistas de aproximadamente la mitad del tamaño.
* Ordenar cada sublistas recursivamente aplicando el ordenamiento por mezcla.
* Mezclar las dos sublistas en una sola lista ordenada.

El ordenamiento por mezcla incorpora dos ideas principales para mejorar su tiempo de ejecución:

* Una lista pequeña necesitará menos pasos para ordenarse que una lista grande.
* Se necesitan menos pasos para construir una lista ordenada a partir de dos listas también ordenadas, que a partir de dos listas desordenadas. Por ejemplo, sólo será necesario entrelazar cada lista una vez que están ordenadas.

La idea de los algoritmos de ordenación por mezcla es dividir la matriz por la mitad una y otra vez hasta que cada pieza tenga solo un elemento de longitud. Luego esos elementos se vuelven a juntar (mezclados) en orden de clasificación.

**Ejemplo:**

Comenzamos dividendo la matriz

**[31,4,88,2,4,2,42]**

**[31,4,88,1] [4,2,42]** Dividimos en 2 partes

**[31,4] [88,1] [4,2] [42]** Dividimos en 4 partes

**[31] [4] [88] [1] [4] [2] [42]** Piezas individuales

Ahora tenemos que unirlos de nuevo en orden de mezcla:

Primero fusionamos elementos individuales en pares. Cada par se fusiona en orden de mezcla.

**[4,31] [1,88] [2,4] [42]**

Luego fusionamos los pares en orden de mezcla:

**[1,4,31,88] [2,4,42]**

Y luego fusionamos los dos últimos grupos.

**[1,2,4,4,31,42,88]**

Y listo, nuestra lista esta ordenada.

4)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Algoritmos de ordenamiento** | **Método burbuja** | **Método inserción** | **Método selección** | **Método QuickSort** | **Método MergeSort** |
| **Breve descripción** | Toma el elemento mayor y recorre X cantidad de veces el arreglo hasta encontrar su lugar. | Es como ordenar un mazo de carta en forma arbitraria. | Consiste en encontrar el menor de todos los elementos y cambiarlo con el de la primera fila. | Utiliza un pivote y ordena los elementos según el pivote elegido. | Divide el arreglo en partes y ordena cada una de ellas. |
| **Característica principal** | Se recorre el arreglo intercambiando los elementos adyacentes. Se recorre el arreglo hasta que no hayan más cambios. | Toma elemento por elemento y avanza hasta su posición con respecto al elemento anterior. | Selecciona el menor elemento de la secuencia no ordenada y lo intercambia. | División por pivote. | Dividir la lista desordenada en varias listas, ordenar cada lista y luego juntar las listas ordenadas. |
| **Ventajas** | \*Fácil implementación.  \*No requiere memoria adicional. | \*Fácil implementación.  \*requiere mínimos requisitos de memoria. | \*Rendimiento constante.  \*Poca diferencia entre el mejor y peor caso. | \*Rápida ejecución.  \*No requiere memoria adicional. | \*Ejecución eficiente. |
| **Desventajas** | \*Realiza numerosas comparaciones.  \*Lento | \*Realiza numerosas comparaciones.  \*Lento | \*Realiza numerosas comparaciones.  \*Lento | \*Trabaja con recursividad  \*Implementación complicada | \*Más uso de memoria ya que divide varias listas  \*Lento |

**Fuentes consultadas**

* Wikipedia
* Khan Academy
* Python diario
* Programación ATS
* Runestone Academy