

**UNIVERSIDAD AUTÓNOMA DE AGUASCALIENTES**

**ESTRUCTURA DE DATOS**

**PROYECTO FINAL, PARTE I**

**TRABAJO DE INVESTIGACIÓN**

**“Métodos de ordenamiento”**

**Nombre del alumno:**

Carlos Daniel Cañedo García

Gilberto Román Rojas Durán

**Grupo:** 3° “B"

**Carrera:** Ingeniería en Sistemas Computacionales

Aguascalientes, Ags. A 16 de diciembre de 2018

1. **Análisis Comparativo métodos de ordenación**

Partimos del hecho de que existen diversos métodos de ordenamiento, mismos que difieren en la manera de aplicarse y la manera en la que pueden crecer o decrecer de acuerdo con la complejidad del algoritmo empleado o de los datos que le han sido introducidos.

Nos vamos a centrar en 7 métodos de ordenamiento: **MergeSort, QuickSort, ShellSort, HeapSort, Selección, Inserción y Burbuja**, para la comparación inicial contamos con una tabla comparativa, considerando un vector de 30,000 elementos, en 3 versiones:

* **Mejor caso:** Se provee un vector en el cual todos los elementos están ordenados de menor a mayor. Los algoritmos lo dejarán intacto.
* **Peor caso:** Se provee un vector con sus datos ordenados de mayor a menor, los métodos de ordenación regresarán el vector totalmente transformado, ninguna posición salvo la de en medio tendrá la misma que antes de iniciar el proceso.
* **Caso promedio:** Se introduce un vector con valores *revueltos*, pueden ser ordenados o no, todo ello se determina de manera aleatoria.

Una vez conocido esto, para los 3 casos se aplicaron los 7 métodos de ordenamiento, mismos que nos arrojaron los siguientes resultados:





Basándonos en los siguientes resultados.

**MergeSort:** En cuanto a número de comparaciones tuvo un comportamiento medio, si bien el número de intercambio de datos se mantuvo igual, el número de comparaciones lógicas aumentó con un caso promedio, entonces hace un consumo mayor de recursos y tiempo cuando se tratan de datos no ordenados, por un algoritmo de crecimiento logarítmico no hubo algún cambio radical en alguno de los 3 casos de prueba. Suponemos que este algoritmo es bueno cuando se tratan listas de un tamaño pequeño, si bien es posible su uso con listas grandes de datos, las excesivas divisiones de las listas y operaciones de intercambio pueden disminuir el rendimiento.

merge sort es un ordenamiento estable, paraleliza mejor, y es más eficiente manejando medios secuenciales de acceso lento. Merge sort es a menudo la mejor opción para ordenar una lista enlazada.

**QuickSort:** Para el mejor y caso promedio tuvimos unos resultados bastante buenos, el número de movimientos de información es mucho más bajo en comparación de todos los métodos de ordenamiento, esto es debido a la elección del pivote y la propia naturaleza del algoritmo, esto quiere decir que si se tiene la certeza de que los datos o están ordenados o están revueltos tendremos una manera óptima de organizar grandes conjuntos de información, para el peor caso tuvimos exponencialmente un número mucho mayor de intercambios, fue uno de los más altos incluso. La implementación de este método es una de las más sencillas realmente, además de que sus resultados generales suelen ser muy buenos salvo el peor caso.

En la práctica, es el algoritmo de ordenación más rápido conocido, su tiempo de ejecución promedio es O(n log (n)), siendo en el peor de los casos O(n2), caso altamente improbable. El hecho de que sea más rápido que otros algoritmos de ordenación con tiempo promedio de O(n log (n)) ( como SmoothSort o HeapSort ) viene dado por que QuickSort realiza menos operaciones ya que el método utilizado es el de partición.

**ShellSort:** Consideramos que el desempeño de este algoritmo para cualquiera de los 3 casos es conservador, siendo para el mejor, peor o caso promedio no crece exponencialmente ni el número de intercambios que se hizo ni el número de comparaciones lógicas, la ejecución consideramos que es de los más rápidos para los casos propuestos, y el número de intercambios de datos es de los más grandes.

El algoritmo ShellSort 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 Shell sort es un simple ordenamiento por inserción, pero para entonces, ya está garantizado que los datos del vector están casi ordenados.

**HeapSort:** Debido a la construcción de un árbol, el número de comparaciones que hace es mucho mayor al de otros algoritmos, al igual que el número de intercambio de datos, el hecho de construir los montículos hace que el tiempo de ejecución de este algoritmo crezca en contraste con los demás, la implementación de este método de ordenamiento no es tan natural como los demás, no es tan asimilable para una persona y la manera en la que se desarrolla es un difícil de darle *traceback*, suponemos que su campo de acción puede ser, para cualquiera de los 3 casos, en conjuntos de datos de tamaños demasiado grandes.

Es un algoritmo de ordenación no recursivo, no estable, con complejidad computacional O (n log n). Basa su funcionamiento en una propiedad de los montículos, por la cual, la cima contiene siempre el menor elemento (o el mayor, según se haya definido el montículo) de todos los almacenados en él.

**Selección:** Al tener una implementación muy secuencial y *muy humana* de realizar, se hace demasiado sencillo de introducir en cualquier algoritmo por su propia sencillez, aunque para el peor, mejor y caso promedio tendrá que hacer las mismas comparaciones, incluso si no tiene nada que intercambiar, por que no partimos de ningún hecho más que el de tener un conjunto de información que necesita ser ordenada, eso hace que se pierda mucho tiempo para cualquiera de los casos, la implementación que recomendamos es para proyectos pequeños, que no estén enfocados a rendimiento ni a aprovechamiento de recursos, si no en ordenamientos simples.

Consiste en encontrar el menor de todos los elementos del arreglo o vector e intercambiarlo con el que está en la primera posición. Luego el segundo mas pequeño, y así sucesivamente hasta ordenarlo todo. Su implementación requiere O(n2) comparaciones e intercambios para ordenar una secuencia de elementos.

**Inserción:** El método que menos permutaciones realiza es el método de inserción en el mejor caso, esto es lógico debido al hecho de que sólo hacemos comparaciones lineales dentro de un ciclo anidado en otro ciclo que va reduciendo su número de iteraciones, esto es muy económico para el uso de recursos **en vectores de un tamaño pequeño y ordenado**, además de ser muy *humano* para ser entendido; sin embargo, si nos vamos al peor caso de este método, vemos que tenemos el número máximo de comparaciones y de intercambio de valores.

Al igual que selección, su implementación por la sencillez es muy fácil, pero para proyectos que necesiten mucho rendimiento no sería recomendada su implementación.

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.

**Burbuja:** Sin optimizaciones, sin separaciones, nada. La ordenación en su concepto más básico se demuestra con este algoritmo, por ello el número de comparaciones lógicas es el más grande en los 3 casos, no importa que ya estén ordenados, es inevitable compararlos. Es el más sencillo de todos, el más entendible, y por ello su implementación sólo debe ser en proyectos pequeños que no demanden complejidad.

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 el más sencillo de implementar.

1. **Cuestionario**
2. **¿Cómo se mido el tiempo de ejecución de un algoritmo?**

Existen diversas maneras en las que se puede medir el tiempo de ejecución de un algoritmo, todas para el mismo fin, pero con un método de empleo distinto. Dichos métodos se explican a continuación:

* **Medir el tiempo de CPU**

Es un método *empírico*, se prueban dos algoritmos que calculan el mismo objeto con datos de distinto tamaño, nos haremos una idea de qué algoritmo es mejor para datos grandes. Para evitar que el resultado tenga en cuenta el efecto de los otros programas que se están ejecutando en nuestro ordenador en ese mismo momento, se usan los conceptos de *CPU time* y *Wall time*, que son los tiempos que el ordenador dedica exclusivamente a nuestro programa.

El *CPU time* es el tiempo de CPU que se ha dedicado a nuestro cálculo, y el *Wall time* el tiempo de reloj entre el comienzo y el final del cálculo. Ambas mediciones son *susceptibles a variaciones imprevisibles*.

* **Contar el número de operaciones**

Una alternativa a medir el tiempo que tarda un programa que implementa un algoritmo es contar directamente el número de operaciones que hace ese algoritmo cuando lo ejecutamos.

**Definición.** Dada una función *g*, diremos que otra función *f* es *O(g)* (y escribiremos f\in O(g) o incluso f=O(g)!), si 0<f(n)<c\:g(n)\qquad\forall n\ge n_0, para constantes positivas c, n_0.

También diremos que f está dominada por g si f\in O(g). Por ejemplo,

n^2 \in O(n^3)

p \in O(\exp)

Donde p es un polinomio cualquiera y \exp es la función exponencial.

1. **¿De qué factores depende el tiempo de ejecución de un algoritmo?**

* **Tamaño o valor de la entrada al programa:** Depende mucho de qué tipo de valor estemos ordenando y la magnitud en tamaño de éste, computacional y temporalmente tiene un menor costo ordenar 10 a 100 datos, para cualquier programa o método de ordenamiento.
* **Calidad del código generado por el compilador:** Cada compilador es un mundo y tiene una manera distinta para trabajar y convertir a lenguaje máquina cada código, incluye optimizaciones propias de cada compilador, es por ello por lo que no siempre dará el mismo resultado el mismo código compilado en cada una de las plataformas.
* **Naturaleza y rapidez de las instrucciones en la máquina utilizada para ejecutar el programa:** Sistema Operativo, CPU, GPU, memorias, almacenamiento, son muchos factores de SW y HW que pueden repercutir en la manera que una PC realiza las operaciones y ejecuta las instrucciones que le han sido dadas, y por supuesto un algoritmo de ordenamiento no se ejecutaría de la misma manera en un ordenador de alta gama con uno convencional.
* **Complejidad en el tiempo del algoritmo representado por el programa:** El propio algoritmo influye mucho en el tiempo de ejecución de él mismo, si se trata de un crecimiento exponencial o logarítmico, si es recursivo, si es iterativo, o por el número de comparaciones que tiene que hacer en cada una de las iteraciones. Todo ello tiene un efecto en el tiempo total de ejecución del programa, y por lo tanto muchas de las veces no llega a ser considerado un factor clave al momento de comparar entre métodos de ordenamiento.

1. **¿Qué es una medida asintótica?**

Al descartar los términos menos significativos y los coeficientes constantes, podemos enfocarnos en la parte importante del tiempo de ejecución de un algoritmo (su tasa de crecimiento) sin involucrarnos en detalles que complican nuestro entendimiento. Cuando descartamos los coeficientes constantes y los términos menos significativos, usamos notación asintótica. Veremos tres formas: notación Θ grande, notación O grande y notación Ω grande.

1. **¿Qué es la cota superior de un algoritmo y con qué símbolo se representa?**

Dada una función *f*, queremos estudiar aquellas funciones *g* que a lo sumo crezcan tan deprisa como *f*. Lo que buscamos es que para una función *g*(*n*), encontrar una función asintótica que cumpla:

O(*g*(*n*)) = {*f*(*n*): con constantes positivas *c* y *n*0 tales que 0 ≤ *f*(*n*) ≤ *cg*(*n*), para toda *n* > *n*0.}

En términos más sencillos, si tenemos una función O(*n*2), quiere decir que el tiempo de ejecución *f*(*n*) nunca será mayor a *cn*2. Como estamos acotando por la parte superior, si un algoritmo cumple O(*n*2), también cumple O(*n*3) (o cualquier otra función de orden mayor), ya que si *f*(*n*) ≤ *cn*2 con mayor razón *f*(*n*) ≤ *cn*3.

Normalmente estamos interesados en la función de menor crecimiento que satisfaga la cota y es común asumir que, en efecto, es la menor que cumple. **Se representa con el símbolo O grande**.

**Sin embargo**, existen otros casos para los cuales se emplea el símbolo Θ y tiene otro significado:

Dada una función *f*, buscamos las funciones *g* que tienen un crecimiento igual a *f*:

Θ(*g*(*n*)) = {*f*(*n*): con constantes positivas *c*1, *c*2 y *n*0 tales que 0 ≤ *c*1*g*(*n*) ≤ *f*(*n*) ≤ *c*2*g*(*n*), para toda *n* > *n*0.}

Lo que se quiere decir con esto, es que a partir de cierto número *n*0 cualquier valor de *f*(*n*) se va a encontrar “empacado” entre *c*1*g*(*n*) y *c*2*g*(*n*). La cota ajustada está cumpliendo al mismo tiempo que la función *g* crezca tan lentamente y tan deprisa como *f*.

Θ(*g*(*n*)) = O(*g*(*n*)) ∩ Ω(*g*(*n*))

Esta notación se acostumbra a utilizarla para acotar el tiempo medio de ejecución de un algoritmo.

1. **¿Qué es la cota inferior de un algoritmo y con qué símbolo se representa?**

Dada una función *f*, queremos estudiar aquellas funciones *g* que crecen tan lentamente como *f*. Esta cota define que, para una función *g*(*n*), tenemos una función asintótica la cual cumple:

Ω(*g*(*n*)) = {*f*(*n*): con constantes positivas *c* y *n*0 tales que 0 ≤ *cg*(*n*) ≤ *f*(*n*), toda *n* > *n*0.}

Esto significa que para valores más grandes que *n*0, los resultados de *f*(*n*) no pueden ser menores a Ω(*n*).

Como esta notación especifica un límite inferior, es común que se utilice para acotar el caso de mejor tiempo de ejecución, ya que cualquier otro caso tiene que ser forzosamente mayor a éste.

Al igual que con la cota superior, la cota inferior que es más útil es aquella que se aproxima en mayor grado a *f* sin sobrepasarla, y estará implícito dentro del libro. También de manera análoga a O, existe una notación similar a Ω que es ω (omega minúscula). En esta cota se cumple que 0 ≤ cg(n) < f(n), y tampoco es utiliza comúnmente. **Se representa con Ω grande.**

1. **¿Qué es la notación O grande?**

La notación O se utiliza la mayoría de las veces para acotar el peor caso de un algoritmo, ya que implica que cualquier otra entrada correrá en un tiempo menor o igual que el especificado por la función.

Algo interesante de esta notación es que en algoritmos iterativos se puede calcular fácilmente contando el mayor número de ciclos anidados de la implementación (aunque no garantiza que sea la menor función). La respuesta más detallada está en el inciso d).

1. **¿A qué se refiere analizar un algoritmo en el mejor, peor y caso promedio?**

En análisis de algoritmos, los términos caso peor, caso mejor y caso promedio tienen los siguientes significados:

* **Caso mejor**: se refiere a la situación inicial de los datos que genera una ejecución del algoritmo con una menor complejidad computacional.
* **Caso peor**: se refiere a la situación inicial de los datos que genera una ejecución del algoritmo con una complejidad computacional mayor.
* **Caso promedio**: la situación inicial de los datos no sigue ningún patrón preestablecido que aporte ventajas o desventajas. Se puede considerar, por tanto, la situación típica de ejecución del algoritmo.

En el caso de los algoritmos de ordenamiento o clasificación, la complejidad computacional está determinada por el número de comparaciones y de asignaciones entre elementos del conjunto que se realiza en una implementación específica del algoritmo.