Preliminares

Los algoritmos que se describen son intercambiables, cada uno con una matriz que contiene los elementos, y asumiendo que todas las posiciones de la matriz contienen datos para ordenar, que "N" es el número de elementos pasados a nuestras rutinas de clasificación y que existen los operadores “<” y “>” (que pueden usarse para realizar un orden consistente en la entrada). Ordenar bajo las condiciones anteriormente descritas se le llama ordenamiento basado en la comparación.
Los parámetros para ordenar representan el marcador de inicio y fin de un contenedor y un comparador opcional, siendo los siguientes:

```
void sort(Iterator begin, Iterator end);
void sort(Iterator begin, Iterator end, Comparator cmp);
```

Los iteradores deben soportar el acceso aleatorio y el algoritmo de clasificación no garantiza que elementos iguales conserven su orden original. Por ejemplo:

```
1- std::sort(v.begin(), v.end());
2- std::sort(v.begin(), v.end(), greater<int>{});
3- std::sort(v.begin(), v.begin() + (v.end() - v.begin()) / 2);
```

1. Ordena todo el contenedor, v, en orden no decreciente.
2. Clasifica todo el contenedor en orden no creciente.
3. Clasifica la primera mitad del contenedor en orden no decreciente.

Insertion sort

El algoritmo

Es uno de los más simples, consiste en N -1 pases. Para pasar p = 1 a N −1, la "insertion sort" garantiza que los elementos en las posiciones 0 a p estén ordenados y aprovecha el hecho de que ya se sabe que los elementos en las posiciones 0 a p − 1 están ordenados.

Original	34	8	64	51	32	21	Positions Moved
After $p = 1$	8	34	64	51	32	21	1
After $p = 2$	8	34	64	51	32	21	0
After $p = 3$	8	34	51	64	32	21	1
After $p = 4$	8	32	34	51	64	21	3
After $p = 5$	8	21	32	34	51	64	4

Figure 7.1 Insertion sort after each pass

En el paso p, se mueve el elemento en la posición p hacia la izquierda hasta encontrar su lugar correcto entre los primeros p + 1 elementos.

Implementación STL de Insertion sort

En lugar de que las rutinas de ordenamiento tomen una serie de elementos comparables como un único parámetro, estas va a recibir un par de iteradores que representan el marcador inicial y final de un rango. Una rutina de ordenamiento de dos parámetros utiliza solo ese par de iteradores y supone que los elementos se pueden ordenar, mientras que una rutina de ordenamiento de tres parámetros tiene un objeto de función como tercer parámetro.

Análisis del Insertion sort

Debido a los bucles anidados, cada uno de los cuales puede tomar "N" iteraciones, el insertion sort es O(N^2). Además, este límite es estricto, porque la entrada en orden inverso puede lograr este límite.

$$\sum_{i=2}^{N} i = 2 + 3 + 4 + \cdots + N = \Theta(N^2)$$

Shellsort

Fue nombrado así gracias a su inventor Donald Shell, fue uno de los primeros algoritmos en romper la barrera del tiempo cuadrático, fue años después de su descubrimiento inicial que se demostró un límite de tiempo sub cuadrático. Funciona comparando elementos que se encuentran distantes, la distancia entre comparaciones disminuye a medida que el algoritmo avanza hasta la última fase, en la que se comparan elementos adyacentes. También se le llama tipo de incremento decreciente.

Original	81	94	11	96	12	35	17	95	28	58	41	75	15
After 5-sort	35	17	11	28	12	41	75	15	96	58	81	94	95
After 3-sort	28	12	11	35	15	41	58	17	94	75	81	96	95
After 1-sort	11	12	15	17	28	35	41	58	75	81	94	95	96

Figure 7.5 Shellsort after each pass

La imagen de arriba muestra una matriz después de varias fases de Shellsort. Una propiedad importante de Shellsort es que un archivo ordenado "hk" que luego está ordenado hk - 1 permanece ordenado "hk". Una opción popular para la secuencia incremental es utilizar la secuencia sugerida por Shell:" ht = [N/2] y hk = [hk+1/2]". El siguiente código contiene una función que implementa "Shellsort".

```
template <typename Comparable>
void shellsort(vector<Comparable> & a)
{
    for(int gap = a.size() / 2; gap > 0; gap /= 2)
        for(int i = gap; i < a.size(); ++i)
        {
            Comparable tmp = std::move(a[i]);
            int j = i;

            for(; j >= gap && tmp < a[j - gap]; j -= gap)
                a[j] = std::move(a[j - gap]);

            a[j] = std::move(tmp);
        }
}
```

Análisis del peor caso de Shellsort

El tiempo de ejecución de "Shellsort" depende de la elección de la secuencia de incremento y las pruebas pueden ser bastante complicadas. El análisis de casos promedio de "Shellsort" es un problema abierto desde hace mucho tiempo, excepto para las secuencias incrementales más triviales.

Heapsort

Heapsort se basa en la idea de que las colas de prioridad para ordenar en tiempo O(N log N), aparte de ofrecer el mejor tiempo de ejecución de Big-Oh. Una estrategia básica es realizar un conjunto de elementos binarios "N", lo cual llevaría un tiempo O(N), para después realizar "N" operaciones de "deleteMin". Se ordenan "N" elementos, al registrar estos elementos en una

segunda matriz y luego copiar la matriz nuevamente; dado que cada "deleteMin" toma O(log N) tiempo, el tiempo de ejecución total es O(N log N).
El problema principal es que utiliza una matriz adicional, por lo tanto se duplica la necesidad de memoria. Una forma de evitar el uso de una segunda matriz es aprovechar el hecho de que después de cada "deleteMin", el "heap" se reduce en 1, por lo tanto, la última celda del "heap" se puede usar para almacenar el elemento que se acaba de eliminar.
Con esta estrategia después del "deleteMin", la matriz contendrá los elementos en orden decreciente. En cambio si se quiere que los elementos estén en el orden creciente más típico, se puede cambiar la propiedad de orden para que el padre tenga un elemento más grande que el hijo.

Análisis de Heapsort

- ***Primera fase:*** constituye la construcción del "heap" y utiliza menos de "2N" comparaciones.
- ***Segunda fase:*** el "ith deleteMax" usa como máximo menos de 2 [log (N - i + 1)] comparaciones, para un total de como máximo 2N logN - O(N) comparaciones (asumiendo N ≥ 2).
  En el peor de los casos, la ordenación en "heap" utiliza como máximo 2 comparaciones N log N − O(N).

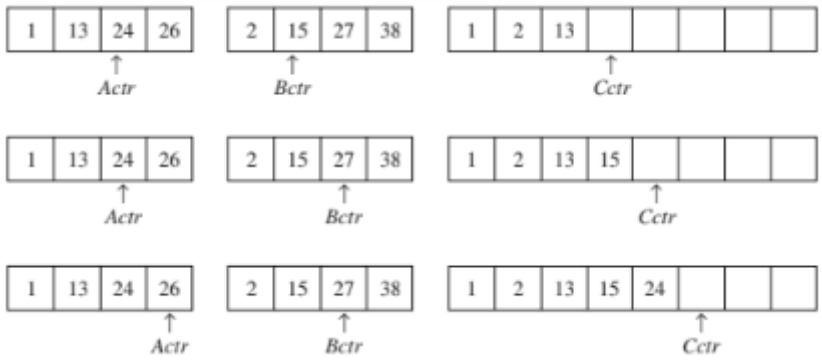
Mergesort

Es un ejemplo de algoritmo recursivo y se ejecuta en el peor de los casos en O(N log N) y el número de comparaciones utilizadas es casi óptimo. La operación fundamental es fusionar dos listas ordenadas lo cual se puede hacer en una pasada por la entrada, si la salida se coloca en una tercera lista.
El algoritmo de toma dos matrices de entrada "A" y "B", una matriz de salida "C" y tres contadores, "Actr", "Bctr" y "Cctr", que inicialmente se configuran al comienzo de sus respectivas matrices; el menor de "A"["Actr"] y "B"["Bctr"] se copia a la siguiente entrada en "C" y avanzan los contadores apropiados; cuando se agota cualquiera de las listas de entrada, el resto de la otra lista se copia a "C".
Un ejemplo de la función explicada es el siguiente:

Si la matriz "A" contiene los números: 1, 13, 24, 26, y "B" contiene 2, 15, 27, 38, entonces el algoritmo procederá de la siguiente manera:

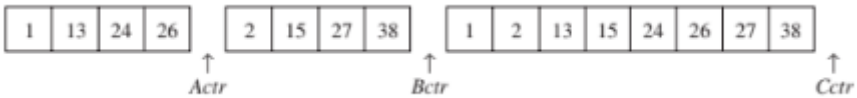
- Primero, hará una comparación entre 1 y 2, 1 se va a agregar a "C" para que después 13 y 2 sean comparados.

- Después, 2 se va a agregar a "C" y 13 se va a comparar con 15. 13 se va a agregar a "C" y 24 y 15 serán comparados. Para luego proceder a seguir realizando esto hasta que 26 y 27 se comparen



- 26 se añade a "C" y la matriz "A" se va a encontrar agotada.

- Finalmente el resto de la matriz "B" será copiada a "C".



El siguiente código es una muestra de implementación de "mergesort", el "mergesort" de un parámetro es solo un controlador para el "mergesort" recursivo de cuatro parámetros:

```
/**
* Algoritmo Mergesort (controlador)
*/
template <typename Comparable>
void mergeSort(vector<Comparable> & a)
{
    vector<Comparable> tmpArray(a.size());

    mergeSort(a, tmpArray, 0, a.size() - 1);
}

/**
* Método interno que hace llamadas reursivas
* "a" es un arreglo de elementos comparables
* "tempArray" es un arreglo que sitúa un resultado ordenado
* "left" es el índice más a la izquierda del subarreglo
* "right" es el índice más a la derecha del subarreglo
*/
template <typename Comparable>
```

```
void mergeSort(vector<Comparable> & a,
                           vector<Comparable> & tmpArray, int left, int right)
{
    if(left < right)
    {
        int center = (left + right) / 2;
        mergeSort(a, tmpArray, left, center);
        mergeSort(a, tmpArray, center + 1, right);
        merge(a, tmpArray, left, center + 1, right);
    }
}
```

Análisis de Mergesort

Es un ejemplo de las técnicas usadas para analizar rutinas recursivas: se tiene que escribir una relación de recurrencia para el tiempo de ejecución. Se va a asumir que "N" es una potencia de 2, de modo que siempre se divide en mitades pares. Para N = 1, el tiempo para fusionar y ordenar es constante, lo que denotaremos por 1, el tiempo para "mergesort" números "N" es igual a el tiempo para hacer dos "mergesort" recursivos del tamaño de N/2 más el tiempo para unir el cual es lineal.

Quicksort

Ha sido el algoritmo de clasificación genérico más rápido conocido en práctica, su tiempo de ejecución promedio es O(N log N). Es muy rápido debido a un bucle interno muy ajustado y altamente optimizado. Tiene un rendimiento O(N^2) en el peor de los casos, pero esto puede volverse exponencialmente improbable. Al combinar Quicksort con Heapsort, podemos lograr el tiempo de ejecución rápido de Quicksort en casi todas las entradas, con el peor tiempo de ejecución de Heapsort O (N log N).
Quicksort es un algoritmo recursivo divide y vencerás. A continuación se va a explicar un algoritmo simple para ordenar una lista.
Se elije de forma arbitraria cualquier item, y después se forman tres grupos:

- Los más pequeños que el item elegido.
- Los que son iguales al item elegido.
- Los que son más largos que el item elegido.
  Después de forma recursiva se ordenan el primer y tercer grupo, para luego concatenar los tres. El resultado será una disposición ordenada de la lista original según los principios básicos de la recursividad.
  Se debe evitar el uso de memoria adicional significativa y tener bucles internos que estén limpios. Por lo tanto, la clasificación rápida comúnmente se escribe de una manera que evita la creación del segundo grupo y el algoritmo tiene numerosos detalles sutiles que

afectan el rendimiento. A continuación la implementación más sencilla de "quicksort" en 4 pasos:

1. Si el número de elementos en "S" es 0 o 1, entonces regresa.
2. Elija cualquier "elemento v" en "S" que será el pivote.
3. Dividir S − {v} (los elementos restantes en S) en dos grupos disjuntos: S1 = {x ∈ S − {v}|x ≤ v}, y S2 = {x ∈ S − {v}|x ≥ v}.
4. Devuelve {quicksort(S1) seguido de v seguido de quicksort(S2)}.
   La razón por la que la "quicksort" es más rápida es que el paso de partición es que se puede realizar en el lugar y de manera muy eficiente lo cual compensa con creces la falta de llamadas recursivas del mismo tamaño.

Escogiendo el pivote

Forma incorrecta: Usar el primer elemento como pivote, lo cual es aceptable si la entrada es aleatoria, pero si la entrada está preclasificada o en orden inverso, entonces el pivote proporciona una partición deficiente.
Maniobra segura: Elegir el pivote al azar lo cual es perfectamente seguro, a menos que el generador de números aleatorios tenga un defecto.
Mediana de tres particiones: La mediana de un grupo de "N" números es el N/2º número más grande, entonces la mejor elección de pivote sería la mediana del conjunto, desafortunadamente, esto es difícil de calcular y ralentizaría considerablemente la clasificación rápida.

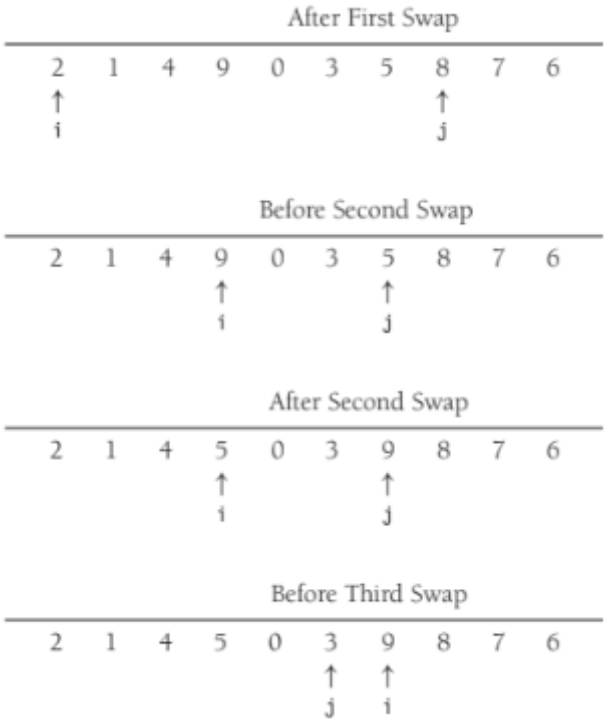
Estrategia de partición

- El primer paso es quitar el elemento pivote cambiándolo por el último elemento. "i" comienza en el primer elemento y "j" comienza en el penúltimo elemento.

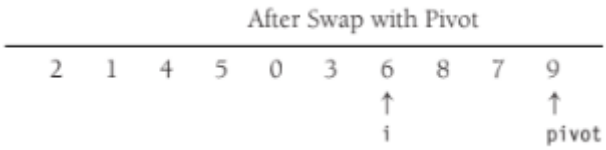
- Lo que nuestra etapa de partición quiere hacer es mover todos los elementos pequeños a la parte izquierda de la matriz y todos los elementos grandes a la parte derecha. Mientras "i" está a la izquierda de "j", se mueve hacia la derecha, omitiendo elementos que son más pequeños que el pivote, por el contrario "j" se mueve hacia la izquierda, omitiendo elementos que son más grandes que el pivote. Cuando "i" y "j" se han detenido, "i" apunta a un elemento grande y "j" apunta a un elemento pequeño. Si "i" está a la izquierda de "j" esos elementos han sido intercambiados, la cuestión es empujar un elemento grande

hacia la derecha y un elemento pequeño hacia la izquierda.

- Luego se intercambian los elementos señalados por "i" y "j" y repetimos el proceso hasta que "i" y "j" se cruzan:



- En esta etapa, "i" y "j" se han cruzado, por lo que no se realiza ningún intercambio. La parte final de la partición es intercambiar el elemento pivote con el elemento señalado por "i":



- Cuando el pivote se intercambia con "i" en el último paso, sabemos que cada elemento en la posición "p" < "i" debe ser pequeño.

Matrices pequeñas

Debido a que quicksort es recursiva no funciona para matrices muy pequeñas (N ≤ 20), e "insertion sort" suele ser mejor. Una solución es no es utilizar la "quicksort" de forma recursiva para matrices pequeñas.

Análisis de quicksort

"Quicksort" es recursivo, para el análisis se asumirá un pivote aleatorio y sin límite para arreglos pequeños. Se va a tomar T(0) = T(1) = 1, el tiempo de ejecución de Quicksort es igual al tiempo de ejecución de las dos llamadas recursivas más el tiempo lineal transcurrido en la partición. Esto da la relación básica de "quicksort":
$T(N) = T(i) + T(N - i - 1) + cN$

Análisis del peor caso

El pivote es el elemento más pequeño, todo el tiempo, entonces i = 0, y si ignoramos T(0) = 1, que es insignificante, la recurrencia es:
$T(N) = T(N - 1) + cN, N > 1$

Análisis del mejor caso

El pivote está en medio. Se asume que los dos sub arreglos tienen cada uno exactamente la mitad del tamaño del original, y aunque esto da una ligera sobreestimación, es aceptable porque solo estamos interesados en una respuesta Big-Oh.
$T(N) = 2T(N/2) + cN$

Análisis de un caso promedio

Se asume que cada uno de los tamaños para "S1" es igualmente probable y, por lo tanto, tiene una probabilidad 1/N. Esto es válido para nuestra estrategia de pivotación y partición, pero no lo es para otras. Las estrategias de partición que no preservan la aleatoriedad de los sub arreglos no pueden utilizar este análisis.

Algoritmo de selección de tiempo esperado lineal

Quicksort puede ser modificado para resolver el problema de selección, al usar una cola de prioridad, podemos encontrar el elemento "k" más grande (o más pequeño) en O(N + k log N). Para el caso especial de encontrar la mediana, esto proporciona un algoritmo O(N log N). El algoritmo "quickselect" puede ordenar la matriz en un tiempo O (N log N), se podría esperar obtener un mejor límite de tiempo para la selección y el algoritmo que se presenta para encontrar el k-ésimo elemento más pequeño en un conjunto S es casi idéntico al de "quicksort":

1. Si |S| = 1, luego k = 1 y devuelve el elemento en S como respuesta. Si se utiliza un límite para matrices pequeñas y |S| ≤ "CUTOFF", luego ordene S y devuelva el k-ésimo elemento más pequeño.
2. Se elije un elemento pivote, $v \in S$.
3. Se divide S - {v} en S1 y S2, como se hizo con la ordenación rápida.
4. Si k ≤ |S1|, entonces el k-ésimo elemento más pequeño debe estar en S1. En este caso, devuelva selección rápida (S1, k). Si k = 1 + |S1|, entonces el pivote es el késimo elemento más pequeño y podemos devolverlo como respuesta. De lo contrario, el k-ésimo elemento más pequeño se encuentra en S2, y es el (k − |S1| − 1)st elemento más pequeño en S2. Hacemos una llamada recursiva y devolvemos selección rápida (S2, k − |S1| − 1).

Ordenamiento de tiempo-lineal: Bucket sort y radix sort

Para que "bucket sort" funcione, debe haber información adicional disponible. La entrada "A1", "A2", ..., "AN" debe consistir únicamente en números enteros positivos menores que "M". Si este es el caso, entonces el algoritmo es simple: se mantiene una matriz llamada recuento, de tamaño "M", que se inicializa completamente a 0. Por lo tanto, el recuento tiene "M" celdas, o depósitos, que inicialmente están vacíos. Cuando se lee "Ai", incrementa el recuento [Ai] en 1. Después de leer toda la entrada, escanea la matriz de recuento e imprime una representación de la lista ordenada. Este algoritmo toma O(M + N); La prueba es dejada como un ejercicio. Si M es O(N), entonces el total es O(N).
La "Radix sort" a veces se conoce como clasificación por tarjetas porque se utilizó hasta la llegada de las computadoras modernas para clasificar tarjetas perforadas de estilo antiguo. Suponiendo que se tienen 10 números en el rango del 0 al 999 que nos gustaría ordenar. En general, se trata de "N" números en el rango de 0 a b p−1 para alguna constante "p". Obviamente no se puede utilizar la clasificación por cubos; habría demasiados cubos. El truco consiste en utilizar varias pasadas de clasificación de cubos. El algoritmo natural sería ordenar por cubos el “dígito” más significativo (el dígito se lleva a la base b), luego el siguiente más significativo, y así sucesivamente. Pero una idea más sencilla es realizar la clasificación de los cubos en orden inverso, empezando por el "dígito" menos significativo.

Ordenamiento externo

Los algoritmos de clasificación externos están diseñados para manejar entradas muy grandes.

¿Por qué se necesitan nuevos algoritmos?

Si la entrada está en una cinta, entonces todas estas operaciones pierden su eficiencia (ya que solo se puede acceder a los elementos de una cinta de forma secuencial), incluso si los datos están en un disco, todavía hay una pérdida práctica de eficiencia debido al retraso necesario para hacer girar el disco y mover su cabeza.

Modelo para ordenamiento externo

Los algoritmos que se consideran funcionan en cintas, son el medio de almacenamiento más restrictivo, dado que el acceso a un elemento en la cinta se realiza enrollando la cinta en la ubicación correcta y solo se puede acceder de manera eficiente a las cintas en orden secuencial.

El algoritmo simple

Usa el algoritmo de fusión de mergesort. Suponiendo que se tienen cuatro cintas, Ta1, Ta2, Tb1, Tb2, que son dos cintas de entrada y dos de salida. Dependiendo del punto del algoritmo,

las cintas a y b son cintas de entrada o cintas de salida. Suponiendo que los datos están inicialmente en Ta1 y además que la memoria interna puede contener (y ordenar) M registros a la vez. Un primer paso natural es leer M registros a la vez desde la cinta de entrada, ordenar los registros internamente y luego escribir los registros ordenados alternativamente en Tb1 y Tb2. Se llama a ejecución a cada conjunto de registros ordenados. Una vez hecho esto, se rebobinan todas las cintas.

Fusión multidireccional

La combinación de dos ejecuciones se realiza enrollando cada cinta de entrada hasta el comienzo de cada ejecución, luego se encuentra el elemento más pequeño, se coloca en una cinta de salida y se avanza la cinta de entrada adecuada. Si hay "k" cintas de entrada, esta estrategia funciona de la misma manera, con la diferencia de que es un poco más complicado encontrar el más pequeño de los "k" elementos. Podemos encontrar el más pequeño de estos elementos utilizando una cola de prioridad. Para obtener el siguiente elemento para escribir en la cinta de salida, se realiza una operación "eliminarMin". Se avanza la cinta de entrada adecuada y, si la ejecución en la cinta de entrada aún no se ha completado, insertamos el nuevo elemento en la cola de prioridad.

Fusión poli fases

La estrategia de fusión requiere el uso de cintas de 2k, lo cual podría resultar prohibitivo para algunas aplicaciones pero es posible arreglárselas con solo cintas k + 1.
Suponiendo que se tienen tres cintas, T1, T2 y T3, y un archivo de entrada en T1 que producirá 34 ejecuciones.

- Una opción es colocar 17 carreras en cada una de las T2 y T3 y luego fusionar este resultado en T1, obteniendo una cinta con 17 ejecuciones.
- El problema es que, dado que todas las ejecuciones están en una cinta y ahora se deben colocar algunas de estas ejecuciones en T2 para realizar otra combinación.
- La forma lógica de hacer esto es copiar las primeras ocho ejecuciones de T1 a T2 y luego realizar la combinación.

Selección de reemplazo

Inicialmente, los registros "M" se leen en la memoria y se colocan en una cola de prioridad. Se realiza un "deleteMin", escribiendo el registro más pequeño (valorado) en la cinta de salida, se lee el siguiente registro de la cinta de entrada.
Si es más grande que el registro que acabamos de escribir, podemos agregarlo a la cola de prioridad. De lo contrario, no podrá pasar a la ejecución actual, dado que la cola de prioridad es un elemento más pequeña, podemos almacenar este nuevo elemento en el espacio muerto de

la cola de prioridad hasta que se complete la ejecución y usar el elemento para la siguiente ejecución.

Investigación

Insertion sort

En el caso óptimo, con los datos ya ordenados, el algoritmo sólo effectura n comparaciones. Por lo tanto la complejidad en el caso óptimo es en Θ(n). En el caso medio, la complejidad de este algoritmo es también en Θ(n2) y en el caso desfavorable, con los datos ordenados a la inversa, se necesita realizar (n-1) + (n-2) + (n-3) .. + 1 comparaciones e intercambios, o (n2-n) / 2. Por lo tanto la complejidad es en Θ(n2).

Shellsort

Existen varias formas de calcular el intervalo, lo cual puede mejorar la efectividad del algoritmo. A continuación, explicaremos la secuencia original propuesta por Shell: n/2, n/4…, n/n, es decir, uno (dividir entre dos, hasta que el último intervalo sea uno), donde n es el tamaño del arreglo.

Quicksort

El método Quicksort es actualmente el mas eficiente y veloz de los método de ordenación interna. Es también conocido con el nombre del método rápido y de ordenamiento por partición. Creado por el científico británico Charles Antony Richard Hoare, tambien conocido como Tony Hoare en 1960.

Bubblesort

Consiste en acomodar el vertor moviendo el mayor hasta la ultima casilla , comenzando desde la casilla cero del vector hasta aver acomodado el número mas grande en la ultima posición, una vez acomodado el mas grande, prosigue a encontrar y acomodar el siguiente más grande comparando de nuevo los números desde el inicio del vector y asi sigue hasta ordenar todos los elementos de todo el arreglo.
Este algoritmo es muy diferente, ya que al ir comparando las casillas para buscar el siguiente mas grande, éste vuelve a comparar las ya ordenadas. A pesar de ser el algoritmo de ordenamiento mas deficiente y el mas usado.

Opinión

Fue un capítulo algo confuso en algunas partes, el más difícil y complejo en mi opinión del libro fue "Quicksort" puesto que los análisis de los casos era algo tedioso y el que le dedicaran varias páginas a este método lo hacía complicado de leer. Y el más simple creo fue el Heapsort

del cual nunca había escuchado pero al leer sobre el se me hizo bastante práctico su proceso y su complejidad no llega a ser cuadrática.

Bibliografía

- “Ordenamiento por inserciÃ³n”. Algorithmique/Programmation. Accedido el 24 de noviembre de 2023. [En línea]. Disponible: http://lwh.free.fr/pages/algo/tri/tri_insertion_es.html#:~:text=El%20ordenamiento%20por%20inserción%20es,cartas%20numeradas%20en%20forma%20arbitraria.
- Ordenamiento por Método Shell”. 403 Forbidden. Accedido el 24 de noviembre de 2023. [En línea]. Disponible: https://repositorio-uapa.cuaieed.unam.mx/repositorio/moodle/pluginfile.php/1472/mod_resource/content/1/contenido/index.html#:~:text=El%20método%20de%20ordenamiento%20Shell,del%20ordenamiento%20de%20inserción%20directa.
- “MÃ©todo Quick Sort”. CIDECAME UAEH. Accedido el 24 de noviembre de 2023. [En línea].
  Disponible: http://cidecame.uaeh.edu.mx/lcc/mapa/proyecto/libro9/mtodo_quick_sort.html