Tema 2: Algoritmos de ordenación, búsqueda y mezcla sobre estructuras de datos no lineales.

Estructuras de Datos No Lineales

Las estructuras de datos no lineales son aquellas en las que cada elemento puede tener varios sucesores y/o predecesores.

Ejemplos:

- Conjuntos dinámicos
- Árboles
- Grafos

Algoritmos de Ordenación

El objetivo principal es obtener algoritmos de ordenamiento lo más eficientes posibles. No existen algoritmos de ordenación con $T(n) < O(\log_2 n)$.

Cada algoritmo estará compuesto de las siguientes operaciones:

- **Comparaciones: que prueban si $A_i < A_j$**
- Intercambios: intercambia el contenido de $A_i \ y \ A_j$
- Asignaciones.

Generalmente, la función de complejidad solo tiene en cuenta las comparaciones.

Los algoritmos de ordenación se clasifican en:

Lentos

Burbuja

Compara e intercambia pares de elementos adyacentes hasta que todos estén ordenados. Compara cada elemento con su sucesor, si no están en orden se intercambian.

El tiempo de ejecución viene determinado por el número de comparaciones. En todos los casos es de O(n^2).

Es un algoritmo simple, sin embargo, solo compara los elementos adyacentes del vector. Lo que lo hace un algoritmo ineficiente.

Inserción

Método de la baraja. Consiste en tomar cada elemento e ir insertándolo en su posición correcta de manera que se mantiene el orden de los elementos ya ordenados.

Inicialmente se toma el primer elemento, a continuación, se toma el segundo y se inserta en la posición adecuada para que ambos estén ordenados. Se toma el tercer elemento y se vuelve a insertar en la posición adecuada para que los tres estén ordenados.

Suponemos el primer elemento ordenado.

Desde el segundo hasta el último elemento:

- Suponer ordenados los primeros (i-1) elementos.

- Toma el elemento i.

- Busca la posición correcta.

- Inserta dicho elemento obteniendo i elementos ordenados.

Complejidad Algoritmo Inserción:

Caso Mejor: Cuando el vector ya está ordenado. Solo se realiza una comparación en cada paso. Para n elementos se hacen n-1 comparaciones, por lo que O(n).

Caso Peor: Cuando el vector está ordenado inversamente. Se realizan n-1 intercambios y n-1 comparaciones, por tanto, T(n) = O(n^2)

Caso Medio: Los elementos aparecen de forma aleatoria. Es la suma de las comparaciones máximas y mínimas dividida entre dos. T(n) = O(n^2).

Selección

Cada vez que se mueve un elemento se lleva a su posición correcta. Se comienza examinando todos los elementos, se localiza el más pequeño y se coloca en la primera posición. A continuación, se localiza el menor de los restantes y se sitúa en la segunda posición. Cuando quedan dos elementos se localiza el menor y se sitúa en la penúltima posición y el último elemento queda colocado en la posición correcta.

Para i desde la primera posición hasta la penúltima hacer:

- localizar el menor desde i hasta el final.

- intercambiar ambos elementos.

El tiempo de ejecución del algoritmo viene determinado por el número de comparaciones, las cuales son independientes del orden original de los elementos. T(n) = O(n^2).

Es muy lenta con vectores grandes y no detecta si el vector está ordenado o parcialmente ordenado.

Intermedios

Shell

Es una mejora de la ordenación por inserción. Se utiliza cuando el número de datos a ordenar es grande.

Para ordenar una secuencia de elementos se procede de la siguiente forma:

1. Se selecciona una **distancia inicial** y se ordenan todos los elementos de acuerdo con esa distancia. Cada elemento separado de otro a *distancia* estará ordenado con respecto a él.
2. Se disminuye esa distancia progresivamente hasta tener distancia 1.

Posición	0	1	2	3	4	5	6	7	8
Original	4	14	21	32	18	17	26	40	6
Distancia 4	4	14	21	32	**6**	17	26	40	**18**
Distancia 2	4	14	**6**	**17**	**18**	**32**	**21**	40	**26**
Distancia 1	4	14	6	17	18	32	21	40	26
Final	4	6	14	17	18	21	26	32	40

Su complejidad en el peor caso es de **O(n^2).** Dependiendo de los incrementos que se usen, la complejidad puede llegar a ser:

- Incrementos de Sell: **O($n^{(3/2)}$)**
- Incrementos de Hibbard: **O($n^{(4/3)}$)**
- Incrementos de Sedgewick: **O($n\ log^2(n)$)**

Rápidos

Merge Sort

Utiliza la estrategia de divide y vencerás.

Primero divide los elementos en dos secuencias de la misma longitud. Después se ordenan los elementos de cada subsecuencia y, por último, se mezclan las dos secuencias para producir la secuencia final ordenada.

Mezcla

Selecciona el menor de los elementos y lo añade a la secuencia final ordenada. Elimina el elemento seleccionado de la secuencia a la que pertenece. Por último, copia en la secuencia final los elementos de la subsecuencia en la que aún quedan elementos.

Teniendo en cuenta que la entrada consiste en **n** elementos y que cada comparación asigna un elemento a un vector auxiliar, el número de comparaciones es n-1 y el número de asignaciones es n. Por tanto, el algoritmo de mezcla se ejecuta en orden lineal (n).

En conclusión, tras resolver la ecuación recursiva que genera el algoritmo, nos da que la complejidad del MergeSort será de $n\,log\,n$.

Quick Sort

Se basa en la estrategia divide y vencerás. Consiste en tomar un valor x como elemento **pivote** y separamos los valores mayores o iguales a x en la derecha y los menores a la izquierda. Después solo tendremos que ordenar los elementos de cada subgrupo.

Para elegir el elemento pivote se pueden seleccionar diferentes formas:

- El primer elemento
- El mayor de los primeros elementos distintos encontrados
- El último elemento
- El elemento medio
- Un elemento aleatorio
- Mediana de tres elementos(primero, medio y último).

El caso mejor del algoritmo de quick-sort se da cuando el pivote divide al conjunto en dos subconjuntos del mismo tamaño. De esta forma se proporciona una complejidad de $T(n) = O(n\,log\,n)$.

Sin embargo, en el caso peor, el algoritmo de quick-sort puede llegar a tener un coste cuadrático en el caso de que el pivote divida el conjunto de elementos en dos conjuntos desiguales (1) y (n-1) elementos.

Heap Sort

Los montículos (heaps) son implementaciones de colas de prioridad. Consiste en un árbol binario con una propiedad invariante: la raíz será menor o igual que el resto de los elementos (montículo de mínimos) o mayor o igual (montículo de máximos).

Para consultar el primer elemento supondrá un coste constante.

Para añadir un elemento, se añade como hoja del árbol y después se la hace **flotar.**

En el peor caso, el elemento añadido se compara con tantos elementos como la altura del árbol O(log n).

Para borrar el elemento mínimo se reemplaza el elemento raíz por una de las hojas y se hunde. O(log n).

Al representarlo en un vector A:

A[2i] será el hijo izquierdo del nodo i.

A[2i+1] será el hijo derecho del nodo i.

A[i/2] será el padre del nodo i.

El coste temporal de este algoritmo será la suma del coste de crear un montículo más el coste de las n-1 llamadas a hundir. En conclusión, el coste será de: $T(n) \in O(n) + (n-1)O(logn) \in O(nlogn)$.

Algoritmos de Búsqueda

Búsqueda Lineal

Dada una clave, se busca en una lista accediendo a todas las posiciones hasta encontrar el elemento o hasta llegar al final de la lista.

En el mejor caso, el elemento se encontrará en la primera posición de la lista. De esta forma tendrá un coste constante.

En el peor de los casos no se encontrará en la lista y habrá que hacer n comparaciones. Esto resulta en un coste lineal.

El caso medio se calcula tomando el tiempo total de encontrar todos los elementos y dividiéndolos por n. Esto resulta en un coste lineal.

Búsqueda Binaria

Busca en una lista **ordenada** un elemento dado. Para ello se sitúa en el centro de la lista y se comprueba si la clave coincide con el valor del elemento central. Si no coincide se compara con él y si es menor descarta el subvector de la derecha, si es mayor descarta el de la derecha. El algoritmo termina cuando encuentra el elemento o cuando el índice izquierdo excede al derecho.

En el peor de los casos tendrá orden logarítmico (O(log n)).

Tablas de Dispersión

Son estructuras de datos diseñadas para implementar diccionarios en un tiempo constante.

La idea fundamental es hacer corresponder a cada clave una posición del espacio de almacenamiento. Debe ser la misma posición siempre y las claves deben de ser independientes unas de otras.

Las tablas hash pueden producir colisiones ya que al aplicar la función de dispersión puede dar la misma clave para dos elementos.

Para resolver el problema de las colisiones existen dos métodos:

- Encadenamiento: Una vez obtenido el índice, se accede a una lista donde se guardan los elementos con la misma clave. En el peor de los casos, todos los elementos estarán en la misma posición y el coste computacional para acceder al último será de n (búsqueda lineal).
  En el caso medio será de coste constante.

- Direccionamiento Abierto: Se busca una posición alternativa utilizando una segunda, tercera… función de dispersión.