**Algoritmo Genérico**

Los Algoritmos Genéticos son un método metaheurístico de optimización estocástica inspirado en la evolución natural de las especies. Establecen una analogía entre el conjunto de soluciones del problema y el conjunto de individuos de una población natural. Así como las poblaciones de individuos evolucionan en cada generación, el conjunto de soluciones mejora en cada iteración.

Los operadores genéticos más utilizados son el *crossover* y la mutación. El *crossover* corresponde al cruzamiento de los individuos de una población y el consiguiente intercambio de información genética en los individuos generados. La mutación, por otra parte, permite introducir la aleatoriedad que puede dar lugar a diferentes soluciones y permiten explorar diferentes zonas de la región factible evitando óptimos locales o convergencia prematura.

Los parámetros del algoritmo genético son el tamaño de la población, el número de generaciones o iteraciones, la probabilidad de crossover y la mutación, estos deben ser determinados por el usuario antes de su ejecución.

**Búsqueda Binaria**

Se comienza de la suposición de que los elementos o datos numéricos se encuentran previamente ordenados y están ubicados en un arreglo, el cual es llamado Espacio de Búsqueda. Primero se ubica el dato en el centro del arreglo, es decir, se toma el arreglo original y se divide en dos subarreglos, para luego seleccionar la casilla que se encuentra entre ambos.

Si la casilla que escogimos es igual al dato que buscamos, hemos encontrado la solución y el algoritmo finaliza. Si ocurre que son distintos, revisamos los dos posibles casos:

1. Si el dato en la casilla es más grande que el dato que estamos buscando, los datos de las casillas mayores también lo serán y podemos desecharlas, puesto que es imposible que el dato buscado se encuentre allí.

2. En caso contrario nos deshacemos de las casillas menores. Cada vez que se realiza lo anterior, el espacio de búsqueda se reduce a la mitad.

Finalmente se aplica el mismo método en las casillas restantes y se repite las veces que sea necesario

Puede que este algoritmo no sea el más rápido a la hora de compararlo con otros que cumplen la misma función, pero es realmente confiable, en parte, debido a su sencillez de código.

La desventaja que representa este método es lo inútil que se vuelve al usarlo si los datos están desordenados, para lo cual es necesario emplear, previamente, algún método de ordenamiento.

**Búsqueda Binaria Recursiva**

Funciona igual que la búsqueda binaria pero sin ciclos iterativos sino con llamadas a si misma.

**Búsqueda Indexada**

**Búsqueda mediante Hashing**

Se refiere a una función o método para generar claves o llaves que representen de manera casi unívoca a un documento, registro, archivo, etc., resumir o identificar un dato a través de la probabilidad, utilizando una función hash o algoritmo hash. Un hash es el resultado de dicha función o algoritmo.

Existen dos formas de Hashing distintas. El primero, llamado Hashing Cerrado, es una estructura de datos estática por lo cual usa un tamaño fijo para el almacenamiento, por lo que limita el tamaño de los conjuntos. El segundo, llamado Hashing Abierto, es una estructura de datos dinámica por lo cual no impone un límite al tamaño del conjunto.

Una función de Hash es una caja negra que tiene como entrada una llave y como salida una dirección.

El hashing es similar al indexamiento en el sentido de asociación entre llaves y direcciones relativas de registros Pero difiere de los índices en 2 cosas:

* La dirección generada por Hash suele ser aleatoria (random). No hay una relación aparente entre la llave y la localización del registro correspondiente.
* El Hash permite que 2 llaves puedan producir la misma salida (direcciones iguales), a esto se le conoce como "colisión".

Idealmente lo que se buscaría es tener un "algoritmo de hash perfecto" en el cual no ocurran colisiones y siempre nos garantice direcciones diferentes. Pero desafortunadamente esto es casi imposible, se dice que 1 de 10^120000 algoritmos evitarían dichas colisiones.

Para reducir el número de colisiones se tienen algunas soluciones: Propagar los registros, usar memoria extra, colocar más de un registro en una dirección.

**Hash Cerrado**

Existen distintos tipos de Hashing cerrados. Los más comunes son el lineal, el doble y el directo, y se diferencian unos de otros por la forma de resolver las colisiones:

* El Hashing lineal resuelve el problema de las colisiones asignando el primer lugar disponible recorriendo circularmente la tabla. Cuando elimina recorre toda la tabla para asegurarse de que lo eliminó. Sufre de agrupamiento primario, dado que cuando existen colisiones la tabla crece por su corrimiento y luego cualquier inserción tendrá colisión.
* El Hashing doble aplica una segunda función Hashing cuando se produce una colisión, para determinar el incremento con el cual buscar la próxima posición. Esta alternativa requiere tablas con números primos de elementos para asegurar el recorrido de todos los lugares posibles.
* El Hashing directo trabaja con un almacén auxiliar, también llamado OVERFLOW, en el cual se almacenan las colisiones.

**Hash Abierto:**

El Hashing abierto maneja las colisiones generando una lista enlazada en cada una de las posiciones de la tabla, es decir, si dos valores poseen un mismo valor para Hashing estarán dentro de una misma lista enlazada.

**Búsqueda en texto**

Los algoritmos de búsquedas de texto en su propósito resuelven el problema de detectar una ocurrencia de una subcadena (llamada patrón o texto de búsqueda, de largo m) en una cadena de caracteres (denominada texto, de largo n).

Existen 3 algoritmos para esta búsqueda y la eficiencia de los algoritmos depende principalmente del largo del patrón.

**Fuerza Bruta**

Consiste en probar todas las posibles posiciones del patrón en el texto.

* Requiere espacio constante.
* Realiza siempre saltos de un carácter.
* Compara de izquierda a derecha.

Ejemplo: Se alinea la primera posición del patrón con la primera posición del texto, y se comparan los caracteres uno a uno hasta que se acabe el patrón, esto es, se encontró una ocurrencia del patrón en el texto, o hasta que se encuentre una discrepancia.

**Knuth Morris-Pratt**

* Para hallar una subcadena en una cadena de texto, este algoritmo utiliza la información obtenida en los fallos anteriores, aprovechando la información que contiene el patrón con el cual se realiza la búsqueda (referente a esto se pre-calcula una tabla). Para de esa manera poder solo iterar una vez la revisión de la cadena de observación.
* Pre-procesa el patrón para asociar a cada carácter que lo forma la longitud del prefijo más largo posible que sea a su vez un sufijo de la cadena de caracteres que precede al patrón y que los caracteres siguientes al prefijo y al sufijo sean diferentes.

Ejemplo:

1. Caso 1: Si (Pj !=Ti and j=0 ), el patrón reanuda las comparaciones un lugar mas adelante.
2. Caso 2: Si (Pj ==Ti and j != m-1 ), Tanto el Ti como el Pj pasan al siguiente índice para volver a comparar.
3. Caso 3: Si (Pj !=Ti and j>0 ), el patrón cambia de lugar en la posición inicial i=i-PP[j-1] y reanuda las comparaciones en donde falló.

**Boyer Moore**

Se considera como el algoritmo de coincidencia de cadena más eficiente en las aplicaciones habituales.

La comparación en este algoritmo se realiza de derecha a izquierda.

**Ordenamiento Quicksort**

* Elegir un elemento de la lista, al que se llama pivote.
* Resituar los otros elementos de la lista, a cada lado del pivote, donde a un lado queden los menores, y al otro lado los mayores.
* La lista queda dividida en dos sub-listas, izquierda y derecha del pivote.
* Se repite este proceso, de manera recursiva, para cada sub-lista generada, mientras ellas contengan más de un elemento.

Este algoritmo posee una fácil implementación lo cual lo hace una buena elección para muchas aplicaciones.

Trabaja muy bien con diferentes tipos de datos de ingreso, aparte de que usa menos recursos que los otros algoritmos de ordenamiento.

**Ordenamiento Margesort**

Este método está basado en la resolución recursiva de un problema dividiéndolo en dos o más sub-problemas de igual tipo o similar. El proceso continúa hasta que éstos llegan a ser lo suficientemente sencillos como para que se resuelvan directamente.



**Heapsort**

Es un método de ordenamiento basado con comparación.

Usa el montículo (Heap) como estructura de datos, el cual representa un árbol.

La ordenación por montículos utiliza la propiedad de la raíz para ordenar el arreglo. Una vez que el arreglo cumpla las propiedades del montículo, quitamos la raíz y la colocamos al final del arreglo. Con los datos que sobran, creamos otro montículo y repetimos hasta que todos los datos estén ordenados.

Las llaves están acomodadas en los nodos de tal manera que, para cada nodo i, Ki <= Kj, donde el nodo j es el padre del nodo i. El árbol se llena de izquierda a derecha.

El ordenamiento por HeapSort realiza los siguientes pasos desde el punto de vista de un Heap (con los elementos) y una lista ordenada (inicialmente vacía):

* Saca el valor máximo del Heap (el que se encuentra en la posición 1)
* Pone el valor que saco del árbol, ahora lo acomoda en el arreglo ordenado.
* Reconstruir el Heap con un elemento menos.

**Shellsort**

Shell Sort es un algoritmo de ordenamiento, basado en comparaciones e intercambios.

si tenemos un arreglo de 10 casillas, nuestros saltos serian:

k = 10/2 = 5

Las primeras comparaciones serian de saltos de 5. 

Una vez que los datos están ordenados, volvemos a calcular el nuevo salto a usar:

K = 5/2 = 2,5 = 2

Ahora las comparaciones serán de saltos de 2. 

Finalmente calculamos el \_ultimo salto:

k = 2/2 = 1

Acá, vemos como el algoritmo se transforma en un algoritmo de inserción directa.



ShellSort inicia una comparación de los elementos más alejados (que distan entre si un intervalo h1).

Luego se van disminuyendo los intervalos, para poder comparar elementos más cercanos, hasta reducirse al método de ordenación por inserción.

En sus primeros saltos trata de desplazar los elementos más chicos al inicio y los más grandes al final.

**Radixsort**

El ordenamiento Radix es un algoritmo que ordena enteros procesando sus dígitos de forma individual. Como los enteros pueden representar cadenas de caracteres (por ejemplo, nombres o fechas).

Existen dos clasi\_caciones de Radix sort: el de dígito menos significativo (LSD) se usa típicamente para ordenar secuencias de enteros como "1, 2, 3, 4, 5, 6, 7, 8, 9, 10 2 el de dígito más significativo (MSD) que trabaja en sentido contrario, se usa frecuentemente para ordenar cadenas de caracteres, como las palabras o representaciones de enteros de longitud fija. Una secuencia como "b, c, d, e, f, g, h, i, j, ba" será ordenada léxicamente como "b, ba, c, d, e, f, g, h, i, j".

**Burbuja Bidireccional**

Los métodos de ordenamiento burbuja son algoritmos de intercambio, es decir, se toman los elementos de dos en dos, se comparan y bajo un determinado criterio se intercambia su valor. Este procedimiento se repite hasta que se comparan todos los elementos y ya no se producen más intercambios.

El método de la burbuja bidireccional representa una mejora real al método de la búrbuja simple, debido a que hace un menor número de comparaciones, más aún cuando el arreglo ya se encuentra ordenado.

**Bucketsort**

Es un algoritmo de ordenamiento que distribuye todos los elementos a ordenar entre un número finito de casilleros. Cada casillero sólo puede contener los elementos que cumplan unas determinadas condiciones. En el ejemplo esas condiciones son intervalos de números. Las condiciones deben ser excluyentes entre sí, para evitar que un elemento pueda ser clasificado en dos casilleros distintos. Después cada uno de esos casilleros se ordena individualmente con otro algoritmo de ordenación (que podría ser distinto según el casillero), o se aplica recursivamente este algoritmo para obtener casilleros con menos elementos.

1. Crear una colección de casilleros vacíos

2. Colocar cada elemento a ordenar en un único casillero

3. Ordenar individualmente cada casillero

4. devolver los elementos de cada casillero concatenados por orden

**Binary tree sort**

Un BST es una estructura de datos abstracta basada en nodos que posee las siguientes características/reglas:

* El subárbol izquierdo solo contiene nodos cuyas llaves son menores a la llave de su raíz.
* El subárbol derecho solo contiene nodos cuyas llaves son mayores a la llave de su raíz.
* Cada subárbol también es un BST.
* No hay llaves duplicadas.

Las principales operaciones de un BST son:

* Inserción
* Borrado: No es necesaria para el sorting.
* Búsqueda: No es necesaria para el sorting.
* Recorridos:
  + EnOrden
  + PostOrden
  + PreOrden