Estudiante: Abner Xocop Chacach
Carné: 20180515
Curso: Estructura de Datos
Profesor: Fernandojose Boiton
Auxiliar: Juan Luis López

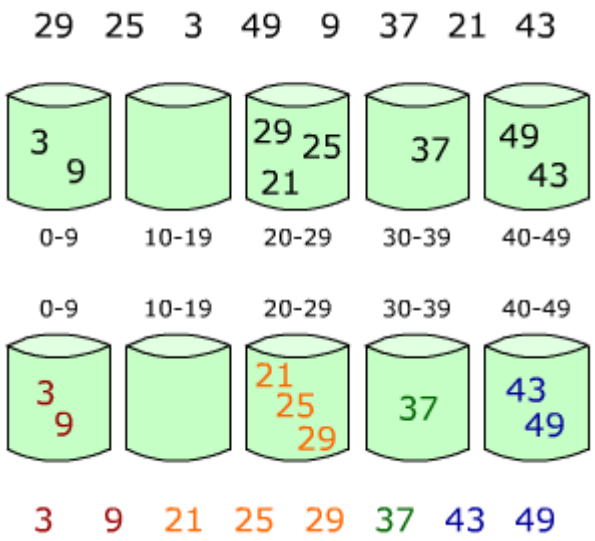
Algoritmos de ordenamiento y búsqueda

Contenido

Algoritmos de ordenamiento

Bucket Sort

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. Se trata de una generalización del algoritmo Pigeonhole sort. ***Cuando los elementos a ordenar están uniformemente distribuidos la complejidad computacional de este algoritmo es de O(n).***



Pseudocódigo

```
función bucket-sort(elementos, n)
  casilleros ← colección de n listas
  para i = 1 hasta longitud(elementos) hacer
    c ← buscar el casillero adecuado
    insertar elementos[i] en casillero[c]
  fin para
  para i = 1 hasta n hacer
    ordenar(casilleros[i])
  fin para
  devolver la concatenación de casilleros[1],..., casilleros[n]
```

Aplicaciones:

Se ha implementado este algoritmo para realizar clasificaciones de grandes cantidades de números celulares según el área, región o compañía telefónica.

Ventajas:

- Es un algoritmo rápido
- Asintóticamente rápido (O(n) cuando la distribución es uniforme)
- Simple de codificar
- Buena para ordenamientos difíciles.

In – place merge sort

Si desea evitar la complejidad de espacio requerida por tener una matriz reutilizable, puede usar el algoritmo de ordenamiento de combinación, pero luego mover los datos dentro de la matriz original. Esta versión admite la formulación recursiva, y el único cambio es en la lógica de fusión.

```
private static void inPlaceSort (Comparable[] x, int first, int
last)
```

El caso base recursivo, como de costumbre, es la matriz de longitud uno (o cero): primero> = último.

De lo contrario, calcular un punto medio (mid = (primera + pasado) / 2;), y después ordenar de forma recursiva los datos de primera a través de mediados, y desde mediados + 1 a través de la última.

inPlaceSort (x, first, mid);

inPlaceSort (x, mid + 1, last);

La parte ligeramente complicada es la lógica de fusión. Como de costumbre, tiene un subíndice en el segmento de matriz izquierdo (x [izquierda]) y el segmento de matriz derecho (x [derecha]). Si x [izquierda] se prueba como menor o igual que x [derecha], entonces ya está en su lugar dentro del segmento del arreglo ordenado, por lo tanto, solo incremente a la izquierda. De lo contrario, el elemento de la matriz en x [derecha] necesita moverse hacia abajo en el espacio actualmente ocupado por x [izquierda], y para acomodarlo, todo el segmento de la matriz desde x [izquierda] hasta x [derecha-1]debe moverse hacia arriba en uno: efectivamente, debe rotar ese pequeño segmento de la matriz. En el proceso, todo se mueve hacia arriba en uno, incluido el final del segmento izquierdo (mediados).

Pseudocódigo

```
left = first;  right = mid+1;
// One extra check:  can we SKIP the merge?
if ( x[mid].compareTo(x[right]) <= 0 )
   return;

while (left <= mid && right <= last)
{  // Select from left:  no change, just advance left
   if ( x[left].compareTo(x[right]) <= 0 )
```

```
            left++;
         // Select from right:  rotate [left..right] and correct
         else
         {  tmp = x[right];     // Will move to [left]
            System.arraycopy(x, left, x, left+1, right-left);
            x[left] = tmp;
            // EVERYTHING has moved up by one
            left++;  mid++;  right++;
         }
      }
      // Whatever remains in [right..last] is in place
```

Aplicaciones:

Hasta el momento, no se ha aplicado a algún caso de la vida real, únicamente se han hecho demostraciones de su eficiencia y para fines educativos.

Complejidad:

O(n^2)

Ventajas:

El comportamiento de tiempo observado de la ordenación in - place es significativamente más rápido que los algoritmos O (n^2) clásicos (selection and insertion), pero aún así (para datos aleatorios e invertidos) puede ajustarse muy bien mediante una línea de tendencia cuadrática dentro de Excel.

Gnome Sort

El algoritmo de ordenación conocido como gNome_sort tiene una historia de invención cuasi paralela. Durante un tiempo existió la polémica sobre su invención, finalmente atribuida a Hamid Sarbazi-Azad quien lo desarrolló en el año 2000 y al que llamó Stupid sort (Ordenamiento estúpido).

Cuando Dick Grune lo inventó (más apropiadamente, lo reinventó) y documentó, no halló evidencias de que existiera y en palabras suyas, dijo de él "the simplest sort algorithm" (es el algoritmo más simple) y quizás tenga razón, pues lo describió en sólo cuatro líneas de código. Dick Grune se basó en los gnomos de jardín holandés y la manera en que se colocan dentro de los maceteros (ver la referencia anterior) y de ahí también el nombre que le dio.

Netamente es un algoritmo de burbuja con una clara particularidad: recorre el array a ordenar como una cremallera, en un vaivén, o bien puede ser definido como un ordenamiento de burbuja bidireccional, que a su vez son llamados también cocktail shaker (agitador de cócteles), por la forma en que trabaja...

Cumple estrictamente hablando con la complejidad O(n^2).

Pseudocódigo:

```
i ← 1
Mientras i ≤ len- 1
  Si i=1 o a[i-1] ≤ a[i]
      i ← i+1
  Sino
      temp ← a[i-1]
      a[i-1] ← a[i]
      a[i] ← temp
      i ← i-1
      Si i = 0
        i ← 1
      Finsi
  Finsi
FinMientras
```

Aplicaciones:

Ninguna, por el momento.

Descripción:

El algoritmo empieza comparando la primera pareja de valores. Si están en orden incrementa el puntero y vuelve a realizar la comparación: si no están en orden, se pasa el menor a la izquierda y el mayor a la derecha, y se reduce el puntero. Ahora la comparación es con el elemento anterior, y si no hay un elemento anterior se pasa al siguiente elemento. Cuando el puntero alcanza el extremo superior del array, ya está totalmente ordenado.

Cuando compara hacia arriba va sin hacer intercambios, es que el par bajo examen está ordenado entre sí, y cuando compara hacia abajo, va haciendo intercambios. El proceso aparece como un zigzagueo continuo a un lado y otro.

La operación empieza por el puntero en el punto más bajo y cuando llega al extremo superior ha terminado de ordenar el array.

Ventajas:

- La operación empieza por el puntero en el punto más bajo y cuando llega al extremo superior ha terminado de ordenar el array.
- Para realizar un ordenamiento inverso basta cambiar la decisión de intercambio de los elementos, es decir, dejar los mayores abajo y los menores arriba.

Algoritmos de búsqueda

Algoritmo de Grover

En computación cuántica, el algoritmo de Grover es un algoritmo cuántico para la búsqueda en una secuencia no ordenada de datos con N componentes en un tiempo O ($N^{1/2}$), y con una necesidad adicional de espacio de almacenamiento de O(logN) (véase notación O). Fue inventado por Lov K. Grover en 1996.

En una búsqueda normal de un dato, si tenemos una secuencia desordenada se debe realizar una inspección lineal, que necesita un tiempo de O (N), por lo que el algoritmo de Grover es una mejora bastante sustancial, evitando, además, la necesidad de la ordenación previa. La ganancia obtenida es "sólo" de la raíz cuadrada, lo que contrasta con otras mejoras de los algoritmos cuánticos que obtienen mejoras de orden exponencial sobre sus contrapartidas clásicas.

Al igual que otros algoritmos de naturaleza cuántica, el algoritmo de Grover es un algoritmo de carácter probabilístico, por lo que produce la respuesta correcta con una determinada probabilidad de error, que, no obstante, puede obtenerse tan baja como se desee por medio de iteraciones.

Aplicaciones:

Aunque el propósito del algoritmo es, como ha sido indicado, la búsqueda en una secuencia se podría describir de una manera más adecuada como la "inversión de una función". Así, si tenemos la función y=f (x), que puede ser evaluada en un computador cuántico, este algoritmo nos permite calcular el valor de x cuando se nos da como entrada el valor de y. Invertir una función puede relacionarse con la búsqueda en una secuencia, si consideramos que la misma es una función que produce el valor de y como la posición ocupada por el valor x en dicha secuencia.

El algoritmo de Grover también se puede utilizar para el cálculo de la media y la mediana de un conjunto de números, y para resolver otros problemas de naturaleza análoga. También se puede utilizar para resolver algunos problemas de naturaleza NP-completa, por medio de inspecciones exhaustivas en un espacio de posibles soluciones. Esto resulta en una apreciable mejora sobre soluciones clásicas.

Complejidad:

O ($N^{1/2}$)

Algoritmo

$$\{|0\rangle, |1\rangle, \cdots, |N-1\rangle\}$$

$$\{\lambda_0, \lambda_1, \cdots, \lambda_{N-1}\}$$

$$U_\omega|\omega\rangle = -|\omega\rangle$$
$$U_\omega|x\rangle = |x\rangle \qquad \text{para todo } x \neq \omega$$
$$\langle\omega|\omega\rangle = 1.$$
$$\langle\omega|x\rangle = \langle x|\omega\rangle = 0\,.$$
$$U_\omega|\omega\rangle = (I - 2|\omega\rangle\langle\omega|)|\omega\rangle = |\omega\rangle - 2|\omega\rangle\langle\omega|\omega\rangle = -|\omega\rangle.$$
$$U_\omega|x\rangle = (I - 2|\omega\rangle\langle\omega|)|x\rangle = |x\rangle - 2|\omega\rangle\langle\omega|x\rangle = |x\rangle.$$

Nuestro objetivo es identificar el autoestado |ω>, o de manera equivalente, el autovalor ω, tal que U_ω

Genetic algorithms

Los algoritmos genéticos (AG) son algoritmos de búsqueda heurística adaptativa que pertenecen a la mayor parte de los algoritmos evolutivos. Los algoritmos genéticos se basan en las ideas de la selección natural y la genética. Se trata de una explotación inteligente de la búsqueda aleatoria provista de datos históricos para dirigir la búsqueda hacia la región de mejor desempeño en el espacio de la solución. Se utilizan comúnmente para generar soluciones de alta calidad para problemas de optimización y búsqueda.

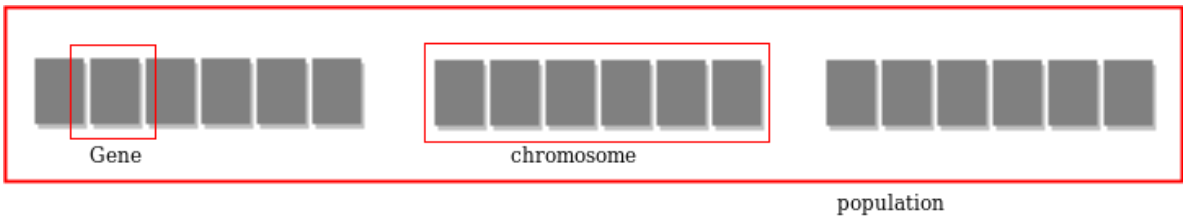
Los algoritmos genéticos simulan el proceso de selección natural, lo que significa que las especies que pueden adaptarse a los cambios en su entorno pueden sobrevivir y reproducirse y pasar a la siguiente generación. En palabras simples, simulan la "supervivencia del más apto" entre individuos de generaciones consecutivas para resolver un problema. Cada generación consta de una población de individuos y cada individuo representa un punto en el espacio de búsqueda y una posible solución. Cada individuo se representa como una cadena de caracteres / integer / float / bits. Esta cadena es análoga al cromosoma.

Los algoritmos genéticos se basan en una analogía con la estructura genética y el comportamiento del cromosoma de la población. La siguiente es la base de las AG basadas en esta analogía:

- Individuo en población compite por recursos y compañero.
- Aquellos individuos que son exitosos (en mejor forma) se aparean para crear más descendientes que otros
- Los genes de los padres "más aptos" se propagan a lo largo de la generación, es decir, a veces los padres crean descendientes, lo que es mejor que cualquiera de los padres.
- Así, cada generación sucesiva es más adecuada para su entorno.

Espacio de búsqueda

La población de individuos se mantiene dentro del espacio de búsqueda. Cada individuo representa una solución en el espacio de búsqueda para un problema dado. Cada individuo se codifica como un vector de longitud finita (análogo al cromosoma) de los componentes. Estas componentes variables son análogas a los genes. Así, un cromosoma (individual) está compuesto de varios genes (componentes variables).



Ternary search

La búsqueda ternaria es un algoritmo de dividir y conquistar que se puede usar para encontrar un elemento en una matriz. Es similar a la búsqueda binaria donde dividimos la matriz en dos partes, pero en este algoritmo. En esto, dividimos la matriz dada en tres partes y determinamos cuál tiene la clave (elemento buscado). Podemos dividir la matriz en tres partes tomando mid1 y mid2, que se pueden calcular como se muestra a continuación. Inicialmente, l y r serán iguales a 0 y n-1 respectivamente, donde n es la longitud de la matriz.

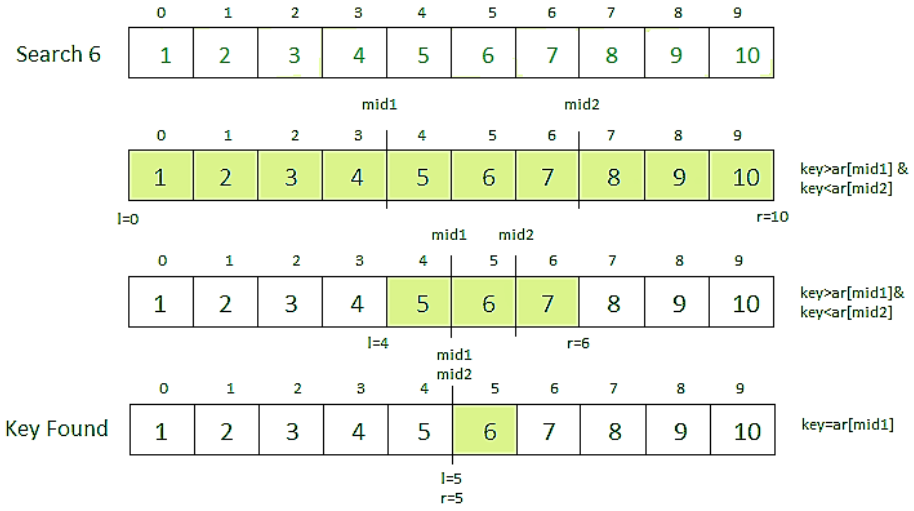
mid1 = l + (rl) / 3
mid2 = r - (rl) / 3

Nota: la matriz debe ordenarse para realizar una búsqueda ternaria en ella.

Pasos para realizar la búsqueda ternaria:

- Primero, comparamos la clave con el elemento en mid1. Si se encuentra igual, devolvemos mid1.
- Si no, comparamos la clave con el elemento en mid2. Si se encuentra igual, devolvemos mid2.
- Si no, verificamos si la clave es menor que el elemento en mid1. Si es así, entonces vuelva a la primera parte.
- Si no, verificamos si la clave es mayor que el elemento en mid2. Si es así, entonces recurrir a la tercera parte.
- Si no, entonces recurrimos a la segunda parte (centro).

Ejemplo:



Complejidad de tiempo*:*

O(log $_3$ n) donde n es el tamaño de la matriz.

Aplicaciones:

En la búsqueda del máximo o mínimo de una función unimodal.

Bonus:

Fibonacci Search

Dada una matriz ordenada arr [] de tamaño n y un elemento x para buscar en ella. Devuelve el índice de x si está presente en la matriz; de lo contrario, devuelve -1.

Ejemplos:

```
Entrada: arr [] = {2, 3, 4, 10, 40}, x = 10
Salida: 3
El elemento x está presente en el índice 3.

Entrada: arr [] = {2, 3, 4, 10, 40}, x = 11
Salida: -1
El elemento x no está presente.
```

Similitudes con la búsqueda binaria:

- Trabajos para matrices ordenadas.
- Un algoritmo de dividir y conquistar.
- Tiene una complejidad de registro y tiempo.

Diferencias con la búsqueda binaria:

- La búsqueda de Fibonacci divide una matriz dada en partes desiguales
- La búsqueda binaria usa el operador de división para dividir el rango. La búsqueda de Fibonacci no usa /, pero usa + y -. El operador de la división puede ser costoso en algunas CPU.
- La búsqueda de Fibonacci examina elementos relativamente más cercanos en pasos subsiguientes. Por lo tanto, cuando la matriz de entrada es grande y no cabe en la memoria caché de la CPU o incluso en la RAM, la búsqueda de Fibonacci puede ser útil.

Algoritmo:

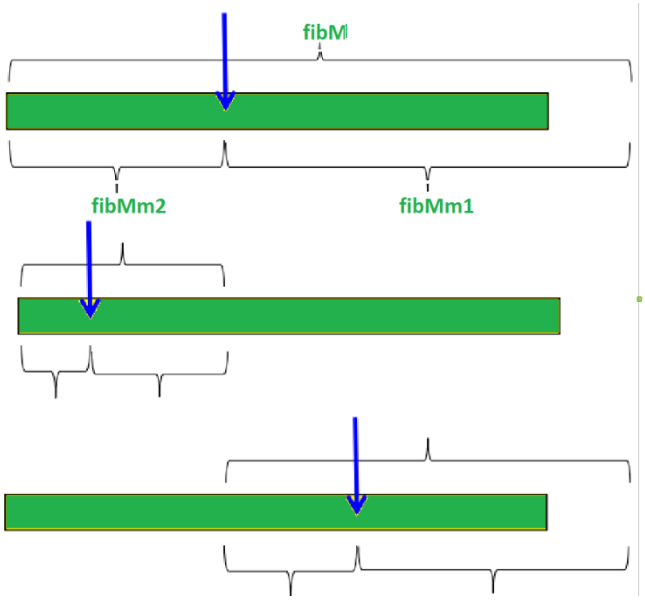
A continuación, se muestra el algoritmo completo

Sea arr [0..n-1] la matriz de entrada y el elemento a buscar sea x.

1. Encuentre el número de Fibonacci más pequeño mayor o igual que n. Deje que este número sea fibM [m'th Fibonacci Number]. Los dos números de Fibonacci que lo preceden son fibMm1 [(m-1) 'th Fibonacci Number] y fibMm2 [(m-2)' th Fibonacci Number].
2. Mientras que la matriz tiene elementos a inspeccionar:
    a. Compara x con el último elemento de la gama cubierta por fibmm2
    b. Si x coincide, devuelve el índice
    c. De lo contrario, si x es menor que el elemento, mueva las tres variables de Fibonacci dos Fibonacci hacia abajo, lo que indica la eliminación de aproximadamente dos tercios posteriores de la matriz restante.
    d. Si x es mayor que el elemento, mueva las tres variables de Fibonacci una Fibonacci hacia abajo. Restablecer la compensación al índice. En conjunto, esto indica la eliminación de aproximadamente un tercio frontal de la matriz restante.
3. Como puede haber un solo elemento restante para comparación, verifique si fibMm1 es 1. Si es Sí, compare x con ese elemento restante. Si coincide, devuelve el índice.

Diagrama

i	1	2	3	4	5	6	7	8	9	10	11	12	13
ar[i]	10	22	35	40	45	50	80	82	85	90	100	-	-



Creación de animaciones

Las siguientes animaciones fueron realizadas en http://www.algomation.com.

- In – place sort:
  - Código GitHub: https://github.com/abnerxch/Estructura-de-datos/tree/master/Sort%20and%20Search%20algorithms/Sort.

- Ternary search:
  - Código GitHub: https://github.com/abnerxch/Estructura-de-datos/tree/master/Sort%20and%20Search%20algorithms/Search.