Cocktail Sort

Bucket Sort

Gnome Sort

Tim Sort

ShellSort

**Circle Sort**

**Gnome Sort**

El algoritmo de ordenación conocido como Gnome\_sort fue inventada por Hamid Sarbazi-Azad, (profesor de la [universidad de Sharif](http://fr.wikipedia.org/wiki/Universit%C3%A9_de_technologie_de_Sharif), una de las mayores universidades de Irán) quien lo desarrolló en el año 2000 y al que llamó Stupid sort (Ordenamiento estúpido).

Cuando [Dick Grune](http://dickgrune.com/index.html) 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, en como se colocan en los maceteros y de ahí también el nombre que le dio.

El algoritmo es similar a la ordenación por inserción , excepto que , en lugar de insertar directamente el elemento a su lugar apropiado , el algoritmo realiza una serie de permutaciones , como en el ordenamiento de burbuja.

void burbuja(int\* vector, int p) {

int i\_b = p;

while ((i\_b>0) && (vector[i\_b]<vector[i\_b - 1])) {

int t = vector[i\_b -1];

vector[i\_b - 1]= vector[i\_b];

vector[i\_b ] = t;

i\_b --;

}

}

void Gnome\_sort (int\* vector) {

for (int i\_i=0;i\_i<20;i\_i++) burbuja(vector,i\_i);

}

TimSort

TimSort es un algoritmo de clasificación basado en la clasificación por inserción y en la clasificación por fusión.

Un algoritmo de ordenamiento estable funciona en tiempo O (n Log n)

Se utiliza en Arrays.sort () de Java, así como en ordenado () y en orden () de Python.

Primero clasifique las piezas pequeñas utilizando la Ordenación por Inserción, luego fusione las piezas utilizando la combinación de la ordenación por fusión.

Dividimos el Array en bloques conocidos como Run. Ordenamos esas ejecuciones utilizando la ordenación de inserción una por una y luego las combinamos utilizando la función de combinación utilizada en la ordenación de combinación. Si el tamaño de la matriz es menor que el de la ejecución, entonces la matriz se clasifica simplemente utilizando el orden de inserción. El tamaño de ejecución puede variar de 32 a 64, dependiendo del tamaño de la matriz. Tenga en cuenta que la función de combinación funciona bien cuando los arreglos secundarios de tamaño son potencias de 2. La idea se basa en el hecho de que la ordenación por inserción funciona bien para arreglos pequeños.

/\* Java program to implement Pigeonhole Sort \*/

import java.lang.\*;

import java.util.\*;

public class GFG

{

public static void pigeonhole\_sort(int arr[],

int n)

{

int min = arr[0];

int max = arr[0];

int range, i, j, index;

for(int a=0; a<n; a++)

{

if(arr[a] > max)

max = arr[a];

if(arr[a] < min)

min = arr[a];

}

range = max - min + 1;

int[] phole = new int[range];

Arrays.fill(phole, 0);

for(i = 0; i<n; i++)

phole[arr[i] - min]++;

index = 0;

for(j = 0; j<range; j++)

while(phole[j]-->0)

arr[index++]=j+min;

}

public static void main(String[] args)

{

GFG sort = new GFG();

int[] arr = {8, 3, 2, 7, 4, 6, 8};

System.out.print("Sorted order is : ");

sort.pigeonhole\_sort(arr,arr.length);

for(int i=0 ; i<arr.length ; i++)

System.out.print(arr[i] + " ");

}

}

// Code contributed by Mohit Gupta\_OMG <(0\_o)>

**Tipo de ciclo**

La clasificación de ciclos es un algoritmo de clasificación in situ, un algoritmo de clasificación inestable, una clasificación de comparación que es teóricamente óptima en términos del número total de escrituras en la matriz original.

Es óptimo en términos de cantidad de escrituras de memoria. Minimiza el número de escrituras de memoria para ordenar (cada valor se escribe cero veces, si ya está en su posición correcta, o se escribe una vez en su posición correcta).

Se basa en la idea de que la matriz a clasificar se puede dividir en ciclos. Los ciclos se pueden visualizar como una gráfica. Tenemos n nodos y un borde dirigido desde el nodo i al nodo j si el elemento en el índice i-th debe estar presente en el índice j-th en la matriz ordenada.

Ciclo en arr [] = {2, 4, 5, 1, 3}

// Java program to implement cycle sort

import java.util.\*;

import java.lang.\*;

class GFG {

// Function sort the array using Cycle sort

public static void cycleSort(int arr[], int n)

{

// count number of memory writes

int writes = 0;

// traverse array elements and put it to on

// the right place

for (int cycle\_start = 0; cycle\_start <= n - 2; cycle\_start++) {

// initialize item as starting point

int item = arr[cycle\_start];

// Find position where we put the item. We basically

// count all smaller elements on right side of item.

int pos = cycle\_start;

for (int i = cycle\_start + 1; i < n; i++)

if (arr[i] < item)

pos++;

// If item is already in correct position

if (pos == cycle\_start)

continue;

// ignore all duplicate elements

while (item == arr[pos])

pos += 1;

// put the item to it's right position

if (pos != cycle\_start) {

int temp = item;

item = arr[pos];

arr[pos] = temp;

writes++;

}

// Rotate rest of the cycle

while (pos != cycle\_start) {

pos = cycle\_start;

// Find position where we put the element

for (int i = cycle\_start + 1; i < n; i++)

if (arr[i] < item)

pos += 1;

// ignore all duplicate elements

while (item == arr[pos])

pos += 1;

// put the item to it's right position

if (item != arr[pos]) {

int temp = item;

item = arr[pos];

arr[pos] = temp;

writes++;

}

}

}

}

// Driver program to test above function

public static void main(String[] args)

{

int arr[] = { 1, 8, 3, 9, 10, 10, 2, 4 };

int n = arr.length;

cycleSort(arr, n);

System.out.println("After sort : ");

for (int i = 0; i < n; i++)

System.out.print(arr[i] + " ");

}

}

// Code Contributed by Mohit Gupta\_OMG <(0\_o)>

Casillero ordenamiento / Pigeonhole Sort

**Peine orden**

Comb Sort es principalmente una mejora sobre Bubble Sort. La ordenación de burbuja siempre compara valores adyacentes. Así que todas las inversiones se eliminan una por una. Comb Sort mejora en Bubble Sort usando un espacio de tamaño mayor que 1. El espacio comienza con un gran valor y se reduce en un factor de 1.3 en cada iteración hasta que alcanza el valor 1. Por lo tanto, Comb Sort elimina más de una cuenta de inversión con una. Intercambia y realiza mejor que Bubble Sort.

Se ha encontrado empíricamente que el factor de reducción es 1.3 (al probar Combsort en más de 200,000 listas aleatorias) [Fuente: Wiki]

Aunque funciona mejor que Bubble Sort en promedio, el peor de los casos sigue siendo O (n2).

A continuación se muestra la implementación.

// Java program for implementation of Comb Sort

**class** CombSort

{

    // To find gap between elements

**int** getNextGap(**int** gap)

    {

        // Shrink gap by Shrink factor

        gap = (gap\*10)/13;

**if** (gap < 1)

**return** 1;

**return** gap;

    }

    // Function to sort arr[] using Comb Sort

**void** sort(**int** arr[])

    {

**int** n = arr.length;

        // initialize gap

**int** gap = n;

        // Initialize swapped as true to make sure that

        // loop runs

**boolean** swapped = **true**;

        // Keep running while gap is more than 1 and last

        // iteration caused a swap

**while** (gap != 1 || swapped == **true**)

        {

            // Find next gap

            gap = getNextGap(gap);

            // Initialize swapped as false so that we can

            // check if swap happened or not

            swapped = **false**;

            // Compare all elements with current gap

**for** (**int** i=0; i<n-gap; i++)

            {

**if** (arr[i] > arr[i+gap])

                {

                    // Swap arr[i] and arr[i+gap]

**int** temp = arr[i];

                    arr[i] = arr[i+gap];

                    arr[i+gap] = temp;

                    // Set swapped

                    swapped = **true**;

                }

            }

        }

    }

    // Driver method

**public** **static** **void** main(String args[])

    {

        CombSort ob = **new** CombSort();

**int** arr[] = {8, 4, 1, 56, 3, -44, 23, -6, 28, 0};

        ob.sort(arr);

        System.out.println("sorted array");

**for** (**int** i=0; i<arr.length; ++i)

            System.out.print(arr[i] + " ");

    }

}

**Tipo de cubo**

La clasificación del cucharón es principalmente útil cuando la entrada se distribuye uniformemente en un rango. Por ejemplo, considere el siguiente problema.

Ordene un gran conjunto de números de punto flotante que estén en el rango de 0.0 a 1.0 y se distribuyan uniformemente en todo el rango. ¿Cómo ordenamos los números de manera eficiente?

Una forma sencilla es aplicar un algoritmo de clasificación basado en comparación. El límite inferior para el algoritmo de ordenación basado en la comparación (clasificación de combinación, clasificación de pila, clasificación rápida, etc.) es Ω (n Log n), es decir, no pueden hacerlo mejor que nLogn.

¿Podemos ordenar la matriz en tiempo lineal? La ordenación de conteo no se puede aplicar aquí ya que usamos claves como índice en la ordenación de conteo. Aquí las claves son números de punto flotante.

#include <stdio.h>

#include <stdlib.h>

**struct** bucket

{

**int** count;

**int**\* value;

};

**int** compareIntegers(**const** **void**\* first, **const** **void**\* second)

{

**int** x = \*((**int**\*)first), y =  \*((**int**\*)second);

**if** (x == y)

    {

**return** 0;

    }

**else** **if** (x < y)

    {

**return** -1;

    }

**else**

    {

**return** 1;

    }

}

**void** bucketSort(**int** **array**[],**int** n)

{

**struct** bucket buckets[3];

**int** i, j, k;

**for** (i = 0; i < 3; i++)

    {

        buckets[i].count = 0;

        buckets[i].value = (**int**\*)**malloc**(**sizeof**(**int**) \* n);

    }

**for** (i = 0; i < n; i++)

    {

**if** (**array**[i] < 0)

        {

            buckets[0].value[buckets[0].count++] = **array**[i];

        }

**else** **if** (**array**[i] > 10)

        {

            buckets[2].value[buckets[2].count++] = **array**[i];

        }

**else**

        {

            buckets[1].value[buckets[1].count++] = **array**[i];

        }

    }

**for** (k = 0, i = 0; i < 3; i++)

    {

*// now using quicksort to sort the elements of buckets*

        qsort(buckets[i].value, buckets[i].count, **sizeof**(**int**), &compareIntegers);

**for** (j = 0; j < buckets[i].count; j++)

        {

**array**[k + j] = buckets[i].value[j];

        }

        k += buckets[i].count;

        free(buckets[i].value);

    }

}

**int** main(**char** \*arg[]) {

**int** **array**[100] = { 5, -34, 10, 1, -42, 123, 2, 395, 5, 4, 1234, 7 };

**int** i = 12,j,k,n;

    n=i;

**printf**("Before Sorting\n");

**for** (j = 0; j<i; j++)

    {

**printf**("%d ", **array**[j]);

    }

    bucketSort(**array**, n);

**printf**("\n After Sorting\n");

**for** (k = 0; k<i; k++)

**printf**("%d ", **array**[k]);

**return** 0;

}

Ordenamiento Shell

Ir a la navegaciónIr a la búsqueda

Proceso paso a paso de ordenamiento según el algoritmo de Shell.

El ordenamiento Shell (Shell sort en inglés) es un algoritmo de ordenamiento. El método se denomina Shell en honor de su inventor Donald Shell. Su implementación original, requiere O(n2) comparaciones e intercambios en el peor caso. Un cambio menor presentado en el libro de V. Pratt produce una implementación con un rendimiento de O(n log2 n) en el peor caso. Esto es mejor que las O(n2) comparaciones requeridas por algoritmos simples pero peor que el óptimo O(n log n). Aunque es fácil desarrollar un sentido intuitivo de cómo funciona este algoritmo, es muy difícil analizar su tiempo de ejecución.

El Shell sort es una generalización del ordenamiento por inserción, teniendo en cuenta dos observaciones:

El ordenamiento por inserción es eficiente si la entrada está "casi ordenada".

El ordenamiento por inserción es ineficiente, en general, porque mueve los valores sólo una posición cada vez.

/\*

\* C Program to sort an array using Shell Sort technique

\*/

#include <stdio.h>

void shellsort(int arr[], int num)

{

int i, j, k, tmp;

for (i = num / 2; i > 0; i = i / 2)

{

for (j = i; j < num; j++)

{

for(k = j - i; k >= 0; k = k - i)

{

if (arr[k+i] >= arr[k])

break;

else

{

tmp = arr[k];

arr[k] = arr[k+i];

arr[k+i] = tmp;

}

}

}

}

}

int main()

{

int arr[30];

int k, num;

printf("Enter total no. of elements : ");

scanf("%d", &num);

printf("\nEnter %d numbers: ", num);

for (k = 0 ; k < num; k++)

{

scanf("%d", &arr[k]);

}

shellsort(arr, num);

printf("\n Sorted array is: ");

for (k = 0; k < num; k++)

printf("%d ", arr[k]);

return 0;

}

<http://lwh.free.fr/pages/algo/tri/tri_es.htm>

<https://es.wikipedia.org/wiki/Gnome_sort>

<https://www.geeksforgeeks.org/timsort/>