LABORATORIO 1

La empresa requiere un programa para ordenar de manera rápida y efectiva una serie de números que el usuario ingrese en la aplicación. La ordenación se puede dar de distintas formas (como el usuario lo indique) y mostrar la secuencia de números ingresados de manera ordenada.

Métodos de ordenamiento:

Ordenamiento burbuja: El funcionamiento se basa en la revisión de cada elemento de la lista que va a ser ordenada con el elemento siguiente, intercambiando sus posiciones si están en el orden equivocado, para esto se requieren varias revisiones hasta que ya no se necesiten más intercambios, lo que muestra que la lista ya ha sido ordenada.

Quick sort: Es el algoritmo más eficiente de todos pues se basa en la técnica de divide y vencerás, que en promedio permite ordenar n elementos en un tiempo proporcional n.Log(n).

Heap sort: Algoritmo de ordenamiento no recursivo, no estable, que consiste en almacenar todos los elementos de un vector a ordenar en un montículo y a extraer el nodo que queda como nodo raíz en sucesivas iteraciones obteniendo el conjunto ordenado.

Merge sort: Se divide en dos procesos; Primero, se divide en partes iguales la lista y el algoritmo que nos permita mezclar los elementos según corresponda.

Shellsort: Los valores de la lista se van casi ordenando y finalmente el último paso es un método por inserción.

Inserción sort: ir insertando un elemento de la lista e ir comparando cada elemento de la parte ordenada con cada elemento de la parte no ordenada.

Selection sort: Consiste en encontrar el menor de todos los elementos en un arreglo e intercambiarlo con el elemento de la primera posición. Luego se halla el segundo elemento más pequeño y así sucesivamente hasta ordenar la totalidad del arreglo.

Complejidad de los tipos de ordenamiento.

Burbuja:

publicstatic int[] OrdenarBurbuja(int[] n){

int temp; 1

int t = n.length; 1

for (int i = 1; i < t; i++) { n+1

for (int k = t- 1; k >= i; k--) { n-i

if(n[k] < n[k-1]){ n

temp = n[k]; 1

n[k] = n[k-1]; 1

n[k-1]= temp; 1

}

}

return n; 1

}

T(n)=

QuickSort:

public static void quicksort(int A[], int izq, int der) {

  int pivote=A[izq];

  int i=izq;

  int j=der;

  int aux;

  while(i<j){

     while(A[i]<=pivote && i<j) i++;

     while(A[j]>pivote) j--;

     if (i<j) {

         aux= A[i];

         A[i]=A[j];

         A[j]=aux;

     }

   }

   A[izq]=A[j];

   A[j]=pivote;

   if(izq<j-1)

      quicksort(A,izq,j-1);

   if(j+1 <der)

      quicksort(A,j+1,der);

}

HeapSort:

public static void heapsort(int n, double ra[]) {   
int l, j, ir, i;   
double rra;   
  
l = (n >> 1) + 1;   
ir = n;   
for (;;) {   
if (l > 1) {   
rra = ra[--l];   
} else {   
rra = ra[ir];   
ra[ir] = ra[1];   
if (--ir == 1) {   
ra[1] = rra;   
return;   
}   
}   
i = l;   
j = l << 1;   
while (j <= ir) {   
if (j < ir && ra[j] < ra[j+1]) { ++j; }   
if (rra < ra[j]) {   
ra[i] = ra[j];   
j += (i = j);   
} else {   
j = ir + 1;   
}   
}   
ra[i] = rra;   
}   
}   
}

MergeSort:

public static void merge(int A[],int izq, int m, int der){

int i, j, k;

int [] B = new int[A.length]; //array auxiliar

for (i=izq; i<=der; i++) //copia ambas mitades en el array auxiliar

B[i]=A[i];

i=izq; j=m+1; k=izq;

while (i<=m && j<=der) //copia el siguiente elemento más grande

if (B[i]<=B[j])

A[k++]=B[i++];

else

A[k++]=B[j++];

while (i<=m) //copia los elementos que quedan de la

A[k++]=B[i++]; //primera mitad (si los hay)

public static void mergesort(int A[],int izq, int der){

if (izq<der){

int m=(izq+der)/2;

mergesort(A,izq, m);

mergesort(A,m+1, der);

merge(A,izq, m, der);

}

}

ShellSort:

public static void shell(int A[]){

int salto, aux, i;

boolean cambios;

for(salto=A.length/2; salto!=0; salto/=2){

cambios=true;

while(cambios){ // Mientras se intercambie algún elemento

cambios=false;

for(i=salto; i< A.length; i++) // se da una pasada

if(A[i-salto]>A[i]){ // y si están desordenados

aux=A[i]; // se reordenan

A[i]=A[i-salto];

A[i-salto]=aux;

cambios=true; // y se marca como cambio.

}

}

}

}

Insertion sort:

public static void insercionDirecta(int A[]){

int p, j;

int aux;

for (p = 1; p < A.length; p++){ // desde el segundo elemento hasta

aux = A[p]; // el final, guardamos el elemento y

j = p - 1; // empezamos a comprobar con el anterior

while ((j >= 0) && (aux < A[j])){ // mientras queden posiciones y el

// valor de aux sea menor que los

A[j + 1] = A[j]; // de la izquierda, se desplaza a

j--; // la derecha

}

A[j + 1] = aux; // colocamos aux en su sitio

}

}

Selection sort:

public static void seleccion(int A[]) {

int i, j, menor, pos, tmp;

for (i = 0; i < A.length - 1; i++) { // tomamos como menor el primero

menor = A[i]; // de los elementos que quedan por ordenar

pos = i; // y guardamos su posición

for (j = i + 1; j < A.length; j++){ // buscamos en el resto

if (A[j] < menor) { // del array algún elemento

menor = A[j]; // menor que el actual

pos = j;

}

}

if (pos != i){

tmp = A[i];

A[i] = A[pos];

A[pos] = tmp;

}

}

}

Criterios:

Criterio A: Complejidad. La complejidad de los algoritmos puede ser:

* [8] complejidad constante.
* [7] complejidad logarítmica.
* [6] complejidad radical.
* [5] complejidad lineal.
* [4] complejidad n(logn).
* [3] complejidad polinómica.
* [2] complejidad exponencial.
* [1] complejidad factorial.

Criterio B: Precisión para resolver el algoritmo.

* [2] Estable.
* [1] No estable.

Criterio C: Eficiencia. Tiempo que tardan en resolver el algoritmo para 1,000,000 de datos por ordenar (1 núcleo, 1GB de RAM).

* [3] Alta (menos de 10 segundos).
* [2] Media (entre 10 y 30 segundos).
* [1] Baja (más de 30 segundos).

Evaluando los algoritmos bajo los anteriores criterios obtenemos la siguiente tabla de calificaciones:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Criterio A | Criterio B | Criterio C | Total |
| Burbuja | 3  O(n^b) | 2  Estable | 1  5584.254499 s | 6 |
| QuikcSort | 3  Promedio: [O](https://es.wikipedia.org/wiki/Cota_superior_asint%C3%B3tica)(n log n), peor caso: O(n²) | 1  inestable | 3  0.291499s | 7 |
| HeapSort | 4  O(n log n) | 1  inestable | 3  0.747395s | 8 |
| MergeSort | 4  O(n log n) | 2  estable | 3  0.704281s | 9 |
| countingsort | 5  O(n+k) | 2  estable | 3  0.016609s | 10 |
| Insertion Sort | 3  O(n^b) | 2  estable | 1  2592.498977s | 6 |
| Selection Sort | 3  O(n^b) | 1  inestable | 1  1935.487457s | 5 |

De acuerdo a los valores de la tabla anterior (evaluación de algunos algoritmos de ordenamiento) se puede establecer que en este caso los mejores algoritmos para ordenar una serie de valores son: countingort, mergeSort y heapSort.