**COMPARACIÓN DE ALGORITMOS DE ORDENAMIENTO**

En este documento se harán la comparación de los siguientes algoritmos de ordenamiento:

Bubble sort

Selection sort

Insertion sort

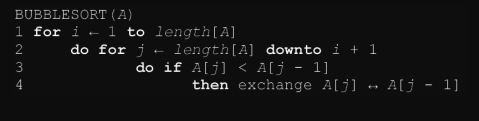
Heap srot

Merge sort

Quick sort

Se probará con un n = 500, 1000, 5000, 10000, 50000, 100000 y 500000, siendo n el tamaño del array.

**Bubble sort**: La complejidad del algoritmo bubble sort es de O(n^2)



void bubble\_sort(int n, int \*arr){

int temp;

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

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

if(arr[j] < arr[i])

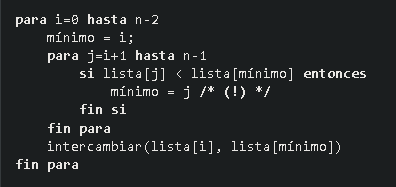
swap(arr[i], arr[j]);

}

}

}

**Selection sort**: La complejidad del algoritmo selection sort es de O(n^2)



void selection\_sort(int n, int \*arr){

int min;

for (int i = 0; i < n-1; ++i){

min = i;

for (int j = i+1; j < n; ++j)

if (arr[j] < arr[min])

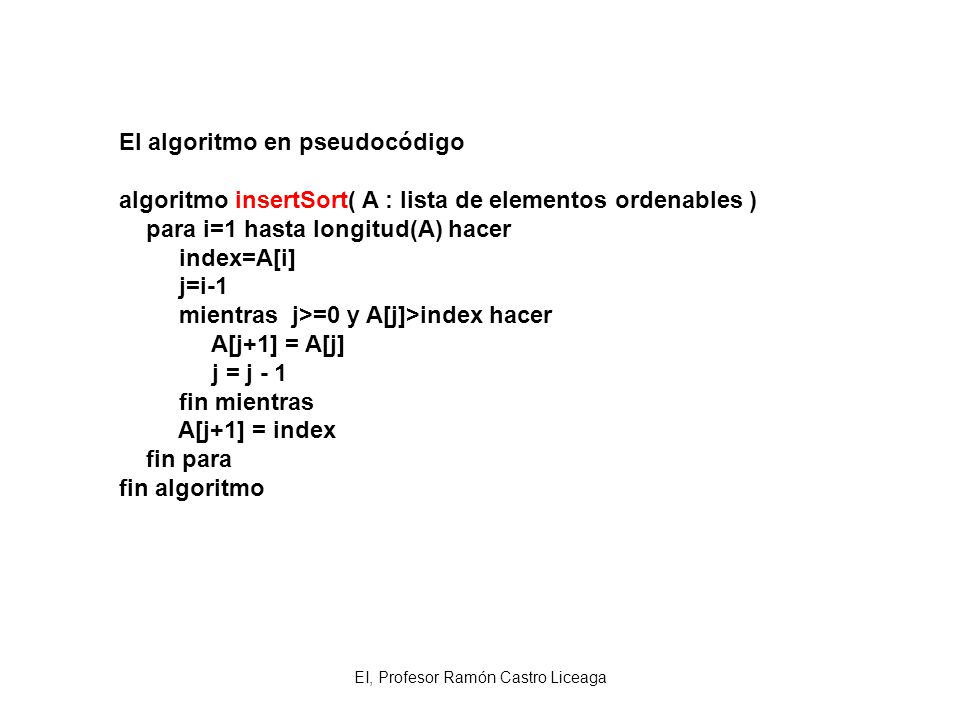
min = j;

swap(arr[min], arr[i]);

}

}

**Insertion sort**: La complejidad del algoritmo selection sort es de O(n^2)



void insertion\_sort(int n, int \*arr){

int key;

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

key = arr[i];

int j = i - 1;

while (j >= 0 and arr[j] > key){

arr[j + 1] = arr[j];

j = j - 1;

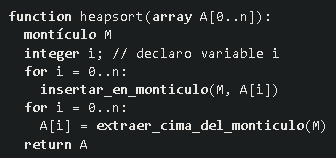
}

arr[j + 1] = key;

}

}

**Heap sort**: Este algoritmo alcanza esta complejidad temporal de O(n log n) debido esencialmente a que utiliza una estructura de datos de modo que cada operación de salida requiere a lo sumo log i pasos, donde i es el número de elementos restantes.



void heapify(int \*arr, int n, int i){

int largest = i;

int l = 2 \* i + 1;

int r = 2 \* i + 2;

if (l < n && arr[l] > arr[largest])

largest = l;

if (r < n && arr[r] > arr[largest])

largest = r;

if (largest != i) {

swap(arr[i], arr[largest]);

heapify(arr, n, largest);

}

}

void heap\_sort(int n, int \*arr){

for (int i = n / 2 - 1; i >= 0; --i)

heapify(arr, n, i);

for (int i = n - 1; i > 0; --i) {

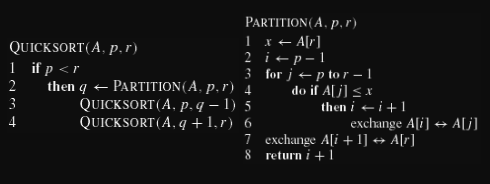
swap(arr[0], arr[i]);

heapify(arr, i, 0);

}

}

**Quick sort**: Quicksort es un algoritmo en O(n^2), sin embargo, en promedio se comporta como un algoritmo O(n logn)



int particion(int \*arreglo, int izquierda, int derecha) {

int pivote = arreglo[izquierda];

while (1) {

while (arreglo[izquierda] < pivote)

izquierda++;

while (arreglo[derecha] > pivote)

derecha--;

if (izquierda >= derecha)

return derecha;

else {

swap(arreglo[izquierda], arreglo[derecha]);

izquierda++;

derecha--;

}

}

}

void quicksort(int \*arreglo, int izquierda, int derecha) {

if (izquierda < derecha) {

int indiceParticion = particion(arreglo, izquierda, derecha);

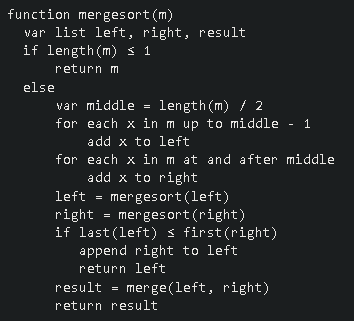
quicksort (arreglo, izquierda, indiceParticion);

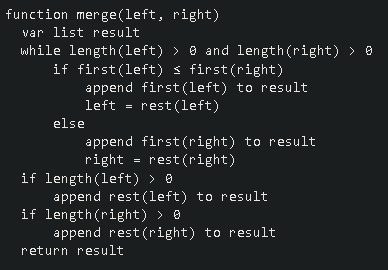
quicksort (arreglo, indiceParticion + 1, derecha);

}

}

**Merge sort**: El algoritmo de ordenamiento por mezcla (merge sort en inglés) es un algoritmo de ordenamiento externo estable basado en la técnica divide y vencerás. Es de complejidad O(n log n).





void merge(int \*array, int const left, int const mid, int const right){

auto const subArrayOne = mid - left + 1;

auto const subArrayTwo = right - mid;

auto \*leftArray = new int[subArrayOne],

\*rightArray = new int[subArrayTwo];

for (auto i = 0; i < subArrayOne; i++)

leftArray[i] = array[left + i];

for (auto j = 0; j < subArrayTwo; j++)

rightArray[j] = array[mid + 1 + j];

auto indexOfSubArrayOne = 0,

indexOfSubArrayTwo = 0;

int indexOfMergedArray = left;

while (indexOfSubArrayOne < subArrayOne && indexOfSubArrayTwo < subArrayTwo) {

if (leftArray[indexOfSubArrayOne] <= rightArray[indexOfSubArrayTwo]) {

array[indexOfMergedArray] = leftArray[indexOfSubArrayOne];

indexOfSubArrayOne++;

}

else {

array[indexOfMergedArray] = rightArray[indexOfSubArrayTwo];

indexOfSubArrayTwo++;

}

indexOfMergedArray++;

}

while (indexOfSubArrayOne < subArrayOne) {

array[indexOfMergedArray] = leftArray[indexOfSubArrayOne];

indexOfSubArrayOne++;

indexOfMergedArray++;

}

while (indexOfSubArrayTwo < subArrayTwo) {

array[indexOfMergedArray] = rightArray[indexOfSubArrayTwo];

indexOfSubArrayTwo++;

indexOfMergedArray++;

}

}

void mergeSort(int \*array, int const begin, int const end){

if (begin >= end)

return;

auto mid = begin + (end - begin) / 2;

mergeSort(array, begin, mid);

mergeSort(array, mid + 1, end);

merge(array, begin, mid, end);

}

**GRAFICCO DE COMPARACION DE TIEMPOS:**

Eje x: tamaño del array

Eje y: tiempo de ejecución

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Tamaño | Bubble | Selection | Insertion | Heap | Quick | Merge |
| 500 | 0.081s | 0.08s | 0.077s | 0.142s | 0.082s | 0.066s |
| 1000 | 0.175s | 0.168s | 0.18s | 0.331s | 0.184s | 0.174s |
| 5000 | 1.001s | 0.92s | 0.96s | 1.732s | 0.954s | 0.869s |
| 10000 | 1.986s | 1.956s | 1.887s | 3.491s | 1.971s | 1.733s |
| 50000 | 13.654s | 14.456s | 12.368s | 17.519s | 9.926s | 8.905s |
| 100000 | 38.149s | 32.948s | 25.122s | 32.881s | 19.164s | 17.671s |
| 5000000 | 457.112s | 458.333s | 273.485s | 149.752s | 80.166s | 77.119s |

Bubble sort

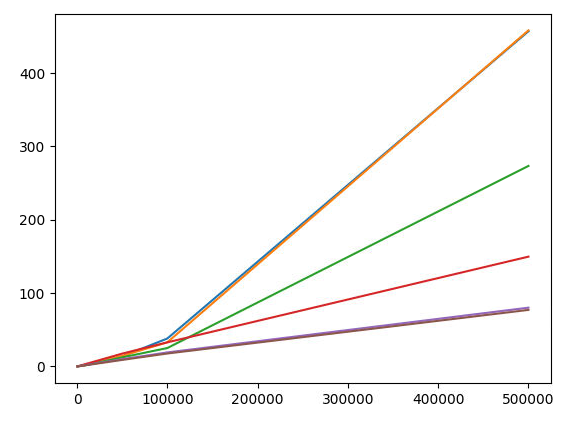
Selection sort

Insertion sort

Heap sort

Quick sort

Merge sort



Como se observa en el primer grafico, el algoritmo quick sort y merge sort son los mas rapidos en tiempo de ejecucion, por otro lado podemos notar que se empiezan a ver los cambios a partir de los 100000 elementos.

