PONTIFICIA UNIVERSIDAD CATÓLICA MADRE Y MAESTRA
FACULTAD DE CIENCIAS DE LA INGENIERÍA
ESCUELA DE INGENIERÍA EN COMPUTACIÓN Y TELECOMUNICACIONES

Asignatura:
Laboratorio de Estructuras de Datos.
Grupo:
ST-ISC-423-171
Periodo Académico:
2020-2021

Práctica #1:
Asignación #1-B

Estudiante:
Junior Hernández 2018-0999

Profesor:
Ing. Jorge A. Luna M.

Fecha de entrega:
6 de noviembre de 2020

Santiago de los caballeros, República Dominicana.

Índice

Introducción

En el siguiente trabajo se presentará y comprobará el funcionamiento de los algoritmos de ordenamiento:

Merge Sort.

Quick Sort.

Radix Sort.

Para así poder confirmar que tanto su aplicación práctica comprueba su definición lógica de su forma teórica, con el objetivo de introducir los algoritmos de ordenamiento y su orden de crecimiento, comportamiento a observarse durante la ejecución de los mismo y realización del experimento todo a través de la tabulación de datos y realización de sus graficas para demostrar lo propuesto por lo anterior.

Metodología de los experimentos

Para realizar este análisis sobre los diversos algoritmos de ordenamiento; Quick Sort, Merge Sort y Radix Sort, ya que se debe de tomar en cuenta con que se está trabajando, las especificaciones de con lo trabajado son:

- ✓ Laptop Dell
- ✓ Sistema Operativo: Windows 10 Home Single Language
- ✓ Procesador: Intel(R) Core(TM) i5-5200U CPU @ 2.20GHz 2.20GHz
- ✓ Memoria instalada (RAM): Sistema operativo de 64 bits, procesador x64
- ✓ Lenguaje de programa: C
- ✓ Compilador: CLion con el IDE por defecto.
- ✓ Microsoft Excel para la representación gráfica

Para generar y tomar el tiempo de trabajo y ejecución de los algoritmos de ordenamiento se procedió a utilizar la librería **#include <time.h>** que viene por defecto, solo teniendo que llamarla y esa generar variables tipo **clock()** o "reloj" en español, para así poder utilizar el siguiente patrón:

```
clock_t start = clock();
//Algoritmos
clock_t stop = clock();
double elapsed = (double) (stop - start) / CLOCKS_PER_SEC;
```

El procedimiento del experimento consistió en dar valores a un puntero con 8 tamaños distintos que van desde 1000 a un 1000000 de elementos, en este caso se eligieron los valores:1000, 5000, 10000, 25000, 50000, 100000, 500000 y 10000000. Para la obtención de resultados un poco más confiables o más exacto, se decidió tomar varias repeticiones para cada uno de los algoritmos y así sacar un promedio de comparaciones, intercambios y sobre todo tiempo. En todo caso para facilitar la hora de obtención de cálculos, se procedió a utilizar la función swap para los intercambios de los arreglos, la función promedioDeArreglo para sacar el promedio de los arreglos y permutation que permite generar valores aleatorios para el arreglo:

```
void swap(int *xp, int *yp)
{
    int temp = *xp;
    *xp = *yp;
    *yp = temp;
}
void permutation(int array[], long size)
{
    srand(time(NULL));
    int i, j;
    for (i = size - 1; i > 0; i--)
    {
        j = rand() % (i + 1);
        swap( &array[i], &array[j]);
    }
}
double promedioDeArreglo(const double arreglo[], int
cantidadDeElementos)
{
    double suma = 0;
    for (int x = 0; x < cantidadDeElementos; x++)
    {
        suma = suma + arreglo[x];
    }
```

```
    return suma / cantidadDeElementos;
}
```

Realización de los experimentos

Generación de tiempos de ejecución

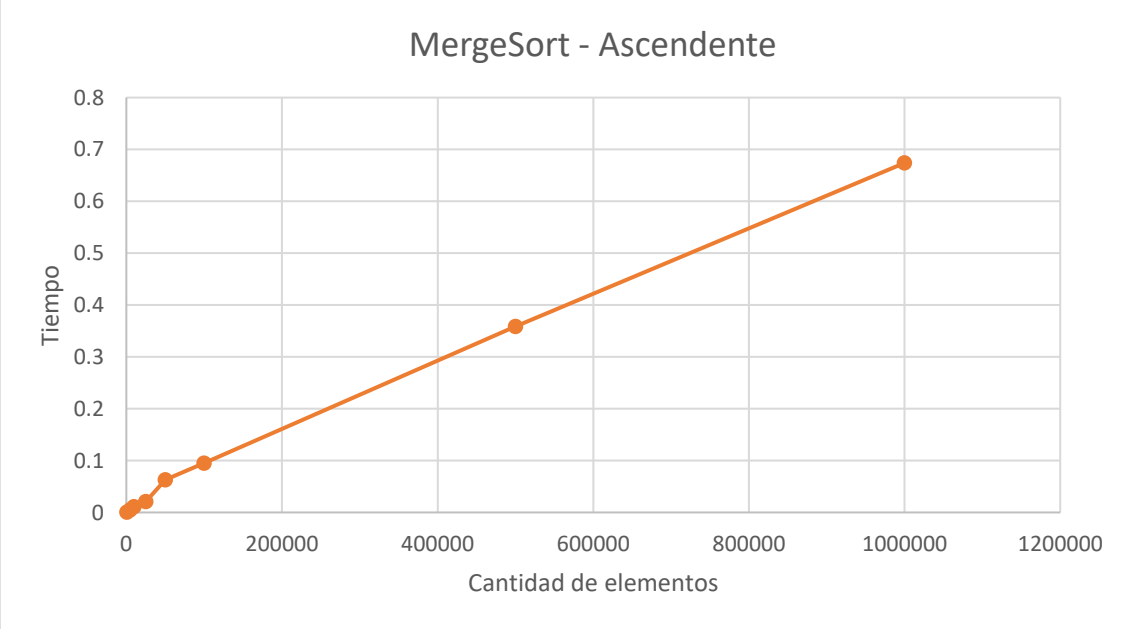
Para promediar el mejor de los casos el arreglo no se mezcló, sino que se mantuvo ordenado y los algoritmos solo lo recorrían deshabilitando la función de `permutation`.

```
for(i = 0; i<n; i++) {
    vector[i] = i+1;
}
//permutation(vector,n);
```

```
void merge(int arr[], int p, int q, int r, int *cont) {

    int n1 = q - p + 1;
    int n2 = r - q;

    int *L = (int *) malloc(n1*sizeof(int));
    int *M = (int *) malloc(n2*sizeof(int));


    for (int i = 0; i < n1; i++){
        L[i] = arr[p + i];
    }
    for (int j = 0; j < n2; j++){
        M[j] = arr[q + 1 + j];
    }

    int i, j, k;
    i = 0;
    j = 0;
    k = p;

    while (i < n1 && j < n2) {
        if (L[i] <= M[j]) {
            arr[k] = L[i];
            i++;
            cont[1]++;
        } else {
            arr[k] = M[j];
            j++;
            cont[1]++;
        }
        k++;
        cont[0]++;
    }

    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }

    while (j < n2) {
```

```
        arr[k] = M[j];
        j++;
        k++;
    }

}

void mergeSort(int arr[], int l, int r, int *cont) {
    if (l < r) {

        int m = l + (r - l) / 2;

        mergeSort(arr, l, m,cont);
        mergeSort(arr, m + 1, r,cont);

        merge(arr, l, m, r,cont);
    }
}
```

Mejor de los casos				
Merge Sort				
Num. Muestra	Num. Elementos	Tiempo de ejecución	Swap elements	Comparation
1	1000	0.001	5044	5044
2	5000	0.005	32004	32004
3	10000	0.011	69008	69008
4	25000	0.021	188476	188476
5	50000	0.063	401952	401952
6	100000	0.095	853904	853904
7	500000	0.359	4783216	4783216
8	1000000	0.674	10066432	10066432



```
void swap2(int *arr,int a,int b)
{
    int t = arr[a];
    arr[a] = arr[b];
    arr[b] = t;
}
```

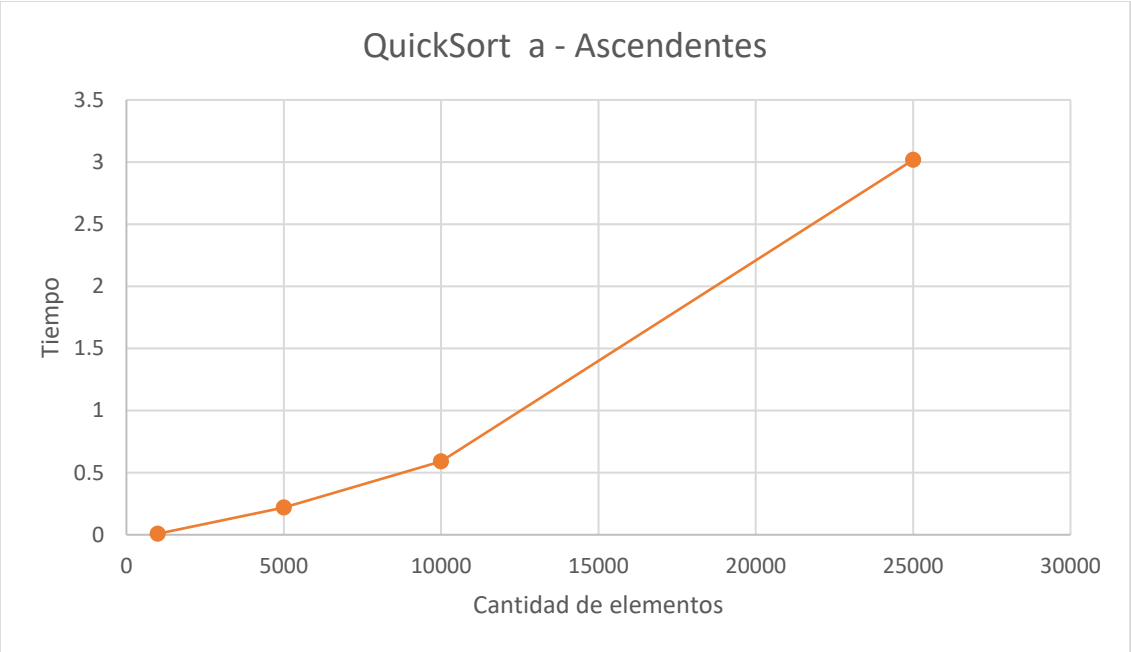
```
void swap(int *a, int *b)
{
    int t = *a;
    *a = *b;
    *b = t;
}
```

a. Para QuickSort: Una versión que siempre utilice el último elemento del arreglo como el pivote.

```
int partition(int *arr, int low, int high,int *cont)
{
    int pivot = arr[high];
    int i = (low - 1);

    for (int j = low; j <= high - 1; j++)
    {
        cont[0]++;
        if (arr[j] <= pivot)
        {
            i++;
            swap2(arr,i,j);
            cont[1]++;
        }
    }
    swap2(arr,i + 1,high);
    cont[1]++;
    return (i + 1);
}
```

```
void QuickSort(int *arr, int low, int high,int *cont)
{
    if (low < high)
    {
        int pi = partition(arr, low, high,cont);

        QuickSort(arr, low, pi - 1,cont);
        QuickSort(arr, pi + 1, high,cont);
    }
}
```

Mejor de los casos				
QuickSort a				
Num. Muestra	Num. Elementos	Tiempo de ejecución	Swap elements	Comparation
1	1000	0.008	500499	499500
2	5000	0.219	12502499	12497500
3	10000	0.591	50004999	49995000
4	25000	3.017	312512499	312487500
5	50000			
6	100000			
7	500000			
8	1000000			

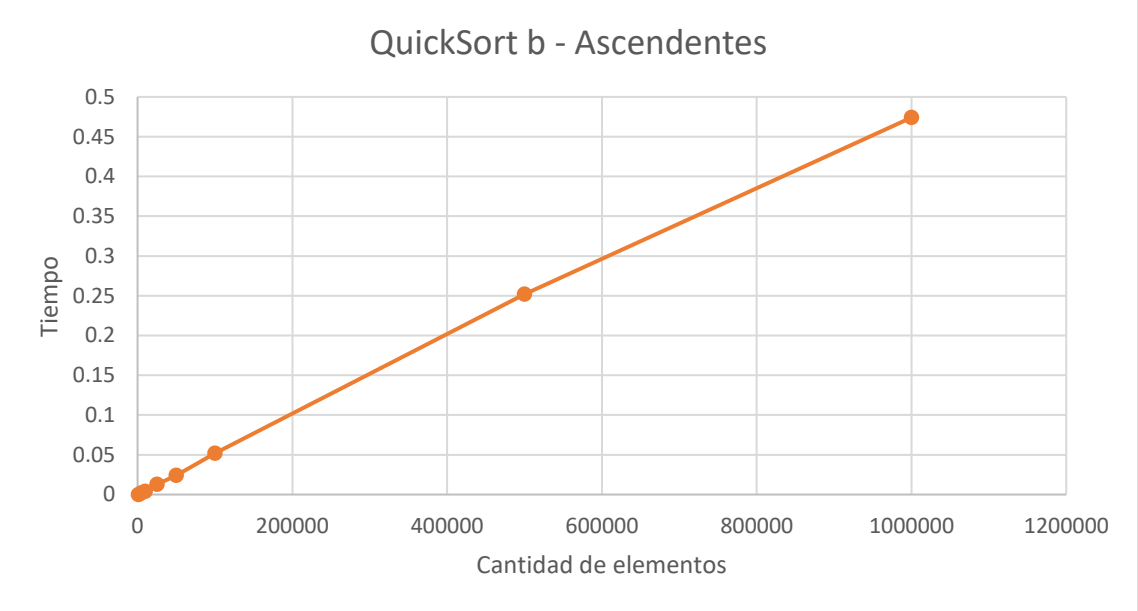


b. Para QuickSort: Una versión que utilice como pivote un elemento aleatorio del mismo.

```
int partition_r(int *arr, int low, int high, int *cont)
{
    srand(time(NULL));
    int random = low + rand() % (high - low);
    swap(&arr[random], &arr[high]);
    cont[1] = cont[1]+1;
    return partition(arr, low, high,cont);
}
```

```
void quickSort(int *arr, int low, int high,int *cont)
{
    if (low < high) {

        int pi = partition_r(arr, low, high,cont);
        quickSort(arr, low, pi - 1,cont);
        quickSort(arr, pi + 1, high,cont);
    }
}
```

	Mejor de los casos			
QuickSort b				
Num. Muestra	Num. Elementos	Tiempo de ejecución	Swap elements	Comparation
1	1000	0	5829	12917
2	5000	0.002	33191	75461
3	10000	0.004	71715	160808
4	25000	0.013	193416	605851
5	50000	0.024	392161	1266438
6	100000	0.052	811902	2845133
7	500000	0.252	4108283	20798123
8	1000000	0.474	8273046	55950287

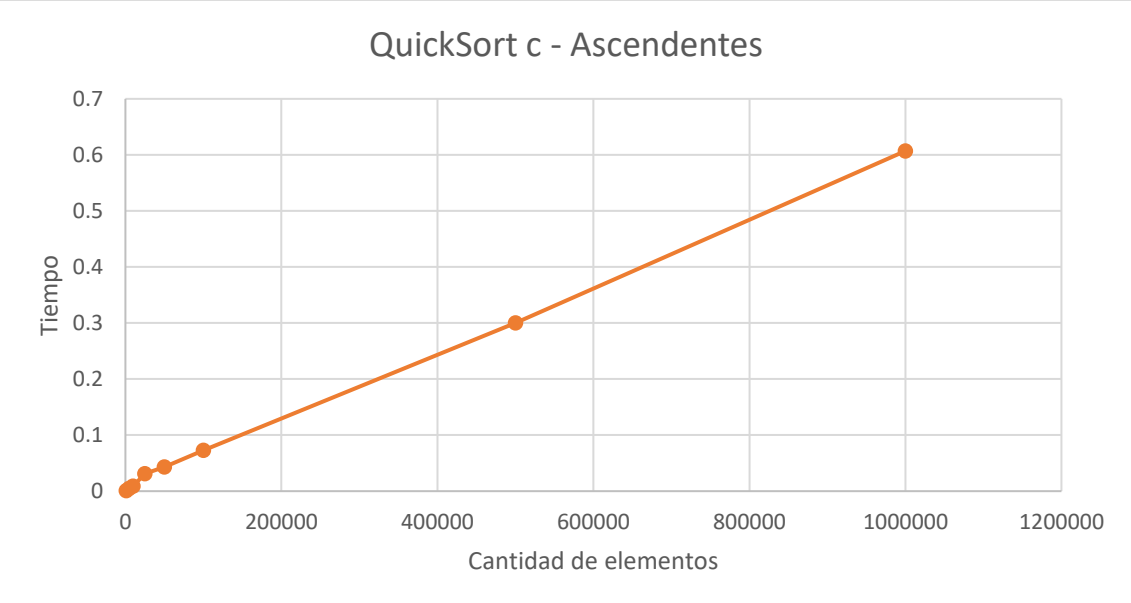


c. Para QuickSort: Una versión que utilice como pivote la media de tres (3) elementos obtenidos de forma aleatoria.
Si existen menos de tres elementos, usted deberá decidir la estrategia del pivote a utilizar.

```
int partition_r_(int *arr, int low, int high,int *cont)
{
    srand(time(NULL));
    int random[3],media;
    for(int i = 0;i < 3;i++)
    {
        random[i] = low + rand() % (high - low);
    }

    media = (random[0] + random[1] +random[2]) / 3;
    swap(&arr[media], &arr[high]);
    cont[1] = cont[1]+1;
    return partition(arr, low, high, cont);
}
```

```c
void Quicksort(int *arr, int low, int high, int *cont)
{
    if (low < high) {

        int pi = partition_r_(arr, low, high,cont);

        Quicksort(arr, low, pi - 1,cont);
        Quicksort(arr, pi + 1, high,cont);
    }
}
```

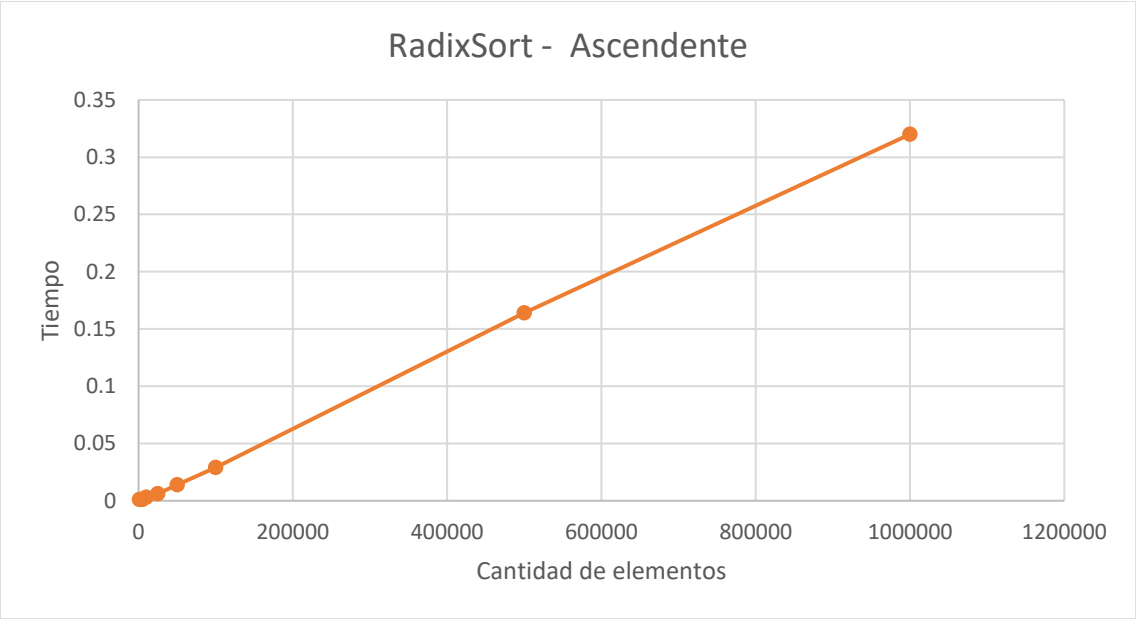
Mejor de los casos				
QuickSort c				
Num. Muestra	Num. Elementos	Tiempo de ejecución	Swap elements	Comparation
1	1000	0.001	4698	9244
2	5000	0.005	31481	56330
3	10000	0.009	70617	123773
4	25000	0.031	178032	350178
5	50000	0.043	382057	740609
6	100000	0.073	781957	1646552
7	500000	0.3	3935108	14079931
8	1000000	0.607	8529033	45174513




```c
int getMax(int array[], int n) {
    int max = array[0];
    for (int i = 1; i < n; i++)
        if (array[i] > max)
            max = array[i];
    return max;
}

void countingSort(int array[], int size, int place, int *cont) {
    int *output;
```

```
    output = (int *) malloc((size+1)*sizeof(int));
    int count[10] = { 0 };

    int max = (array[0] / place) % 10;

    for (int i = 1; i < size; i++) {
        cont[0]++;
        if (((array[i] / place) % 10) > max) {
            max = array[i];
            cont[1]++;
        }
    }

    count[max + 1];

    //for (int i = 0; i < max; ++i)
      //  count[i] = 0;

    for (int i = 0; i < size; i++)
        count[(array[i] / place) % 10]++;

    for (int i = 1; i < 10; i++)
        count[i] += count[i - 1];

    for (int i = size - 1; i >= 0; i--) {
        output[count[(array[i] / place) % 10] - 1] = array[i];
        cont[1]++;
        count[(array[i] / place) % 10]--;
    }

    for (int i = 0; i < size; i++){
        array[i] = output[i];
        cont[1]++;
    }

}

void radixsort(int array[], int size, int * cont) {
    int max = getMax(array, size);

    for (int place = 1; max / place > 0; place *= 10)
        countingSort(array, size, place, cont);
}
```

Mejor de los casos				
RadixSort				
Num. Muestra	Num. Elementos	Tiempo de ejecución	Swap elements	Comparation
1	1000	0.001	6004	2994
2	5000	0.001	40003	19992
3	10000	0.003	80003	39992
4	25000	0.006	250002	124990
5	50000	0.014	500002	249990
6	100000	0.029	1000002	499990
7	500000	0.164	6000001	2999988
8	1000000	0.32	12000001	5999988



Tiempos de ejecución en casos intermedios

Para este si se mezclaron los elementos en el arreglo de forma aleatoria para que los algoritmos los organizaran de manera ascendente habilitando la función de permutaciones.

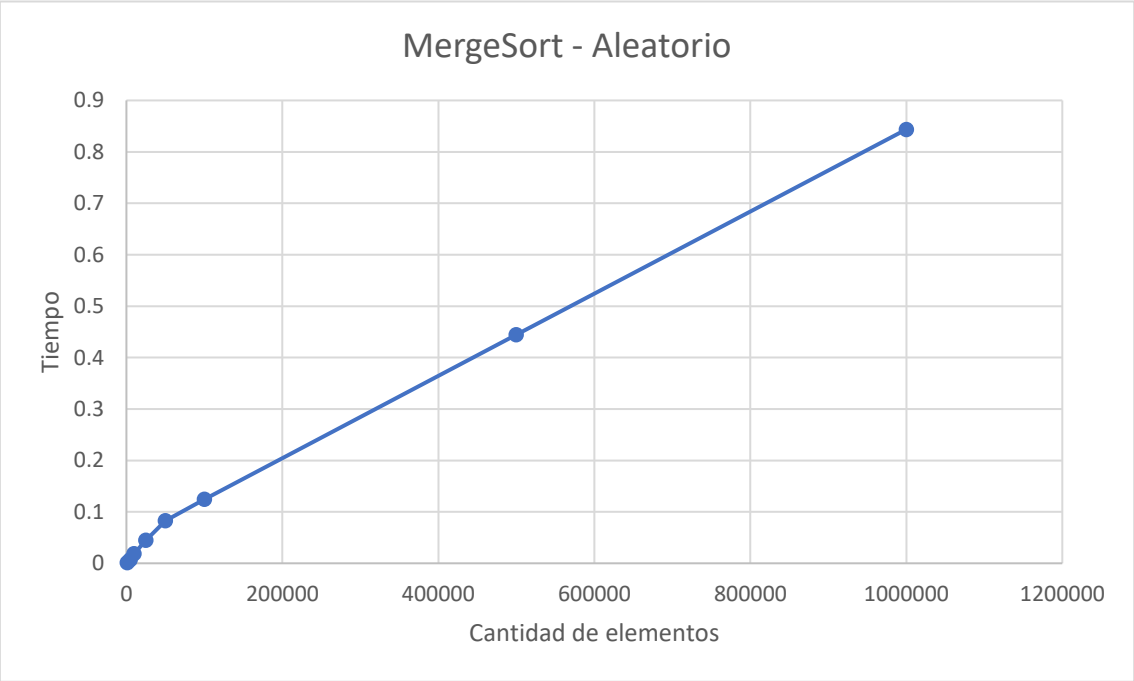
```
for (i = 0; i < n; i++){
    vector[i] = i + 1;
}
permutation(vector,n);
```

Prueba No.1 -Aleatoria				
MergeSort				
Num. Muestra	Num. Elementos	Tiempo de ejecución	Swap elements	Comparation
1	1000	0.001	8732	8732
2	5000	0.007	55199	55199
3	10000	0.014	120419	120419
4	25000	0.04	334069	334069
5	50000	0.072	718223	718223
6	100000	0.089	1536631	1536631
7	500000	0.433	8835709	8835709
8	1000000	0.805	18566620	18566620

Prueba No.2 -Aleatoria				
MergeSort				
Num. Muestra	Num. Elementos	Tiempo de ejecución	Swap elements	Comparation
1	1000	0.001	8740	8740
2	5000	0.006	55303	55303
3	10000	0.028	120536	120536
4	25000	0.065	334080	334080
5	50000	0.128	718110	718110
6	100000	0.158	1535981	1535981
7	500000	0.433	8836403	8836403
8	1000000	0.765	18509125	18509125

Prueba No.3 -Aleatoria				
MergeSort				
Num. Muestra	Num. Elementos	Tiempo de ejecución	Swap elements	Comparation
1	1000	0.001	8722	8722
2	5000	0.007	55248	55248
3	10000	0.013	120344	120344
4	25000	0.029	334123	334123
5	50000	0.048	718235	718235
6	100000	0.127	1536032	1536032
7	500000	0.468	8832258	8832258
8	1000000	0.961	18510722	18510722

Promedio -Aleatoria				
MergeSort				
Num. Muestra	Num. Elementos	Tiempo de ejecución	Swap elements	Comparation
1	1000	0.001	8731.333333	8731.33333
2	5000	0.006666667	55250	55250
3	10000	0.018333333	120433	120433
4	25000	0.044666667	334090.6667	334090.667
5	50000	0.082666667	718189.3333	718189.333
6	100000	0.124666667	1536214.667	1536214.67
7	500000	0.444666667	8834790	8834790
8	1000000	0.843666667	18528822.33	18528822.3



a. Para QuickSort: Una versión que siempre utilice el último elemento del arreglo como el pivote.

Prueba No.1 -Aleatoria				
QuickSort a				
Num. Muestra	Num. Elementos	Tiempo de ejecución	Swap elements	Comparation
1	1000	0.001	6724	11570
2	5000	0.001	42518	80596
3	10000	0.003	84621	151432
4	25000	0.008	268809	452027
5	50000	0.011	534071	946947
6	100000	0.026	1010118	2057809
7	500000	0.133	5469962	13047712
8	1000000	0.257	11308768	35262917

Prueba No.2 -Aleatoria				
QuickSort a				
Num. Muestra	Num. Elementos	Tiempo de ejecución	Swap elements	Comparation
1	1000	0	6030	10390
2	5000	0.001	37776	73446
3	10000	0.002	76905	146908
4	25000	0.007	218648	410216
5	50000	0.019	472287	895705
6	100000	0.033	1060796	2020663
7	500000	0.123	5326922	13141179
8	1000000	0.268	10998258	35408676

Prueba No.3 -Aleatoria				
QuickSort a				
Num. Muestra	Num. Elementos	Tiempo de ejecución	Swap elements	Comparation
1	1000	0	7422	11990
2	5000	0.001	34376	67220
3	10000	0.002	81481	148298
4	25000	0.005	241513	442524
5	50000	0.011	484476	925345
6	100000	0.043	989992	2033689
7	500000	0.149	5374410	15812978
8	1000000	0.248	10732115	34293508

Promedio -Aleatoria				
QuickSort a				
Num. Muestra	Num. Elementos	Tiempo de ejecución	Swap elements	Comparation
1	1000	0.00033333	6725.333333	11316.66667
2	5000	0.001	38223.33333	73754
3	10000	0.00233333	81002.33333	148879.3333
4	25000	0.00666667	242990	434922.3333
5	50000	0.01366667	496944.6667	922665.6667
6	100000	0.034	1020302	2037387
7	500000	0.135	5390431.333	14000623
8	1000000	0.25766667	11013047	34988367

b. Para QuickSort: Una versión que utilice como pivote un elemento aleatorio del mismo.

Prueba No.1 -Aleatoria				
QuickSort b				
Num. Muestra	Num. Elementos	Tiempo de ejecución	Swap elements	Comparation
1	1000	0.001	6959	11336
2	5000	0.004	43092	68682
3	10000	0.03	98223	160996
4	25000	0.026	236233	454586
5	50000	0.033	587333	966537
6	100000	0.075	1184459	2054542
7	500000	0.32	5750347	14103258
8	1000000	0.606	11586201	31516715

Prueba No.2 -Aleatoria				
QuickSort b				
Num. Muestra	Num. Elementos	Tiempo de ejecución	Swap elements	Comparation
1	1000	0	7309	11952
2	5000	0.004	43710	70541
3	10000	0.006	84403	154967
4	25000	0.034	283953	450262
5	50000	0.035	576476	989112
6	100000	0.076	1215850	1982492
7	500000	0.271	5954626	11978170
8	1000000	0.542	11874327	27049192

Prueba No.3 -Aleatoria				
QuickSort b				
Num. Muestra	Num. Elementos	Tiempo de ejecución	Swap elements	Comparation
1	1000	0	8273	11847
2	5000	0.003	45332	75353
3	10000	0.006	89446	159223
4	25000	0.017	232091	417629
5	50000	0.034	498383	934234
6	100000	0.063	1130864	2027359
7	500000	0.228	6044755	12266115
8	1000000	0.484	11918696	29599417

Promedio -Aleatoria				
QuickSort b				
Num. Muestra	Num. Elementos	Tiempo de ejecución	Swap elements	Comparation
1	1000	0.00033333	7513.666667	11711.6667
2	5000	0.00366667	44044.66667	71525.3333
3	10000	0.014	90690.66667	158395.333
4	25000	0.02566667	250759	440825.667
5	50000	0.034	554064	963294.333
6	100000	0.07133333	1177057.667	2021464.33
7	500000	0.273	5916576	12782514.3
8	1000000	0.544	11793074.67	29388441.3

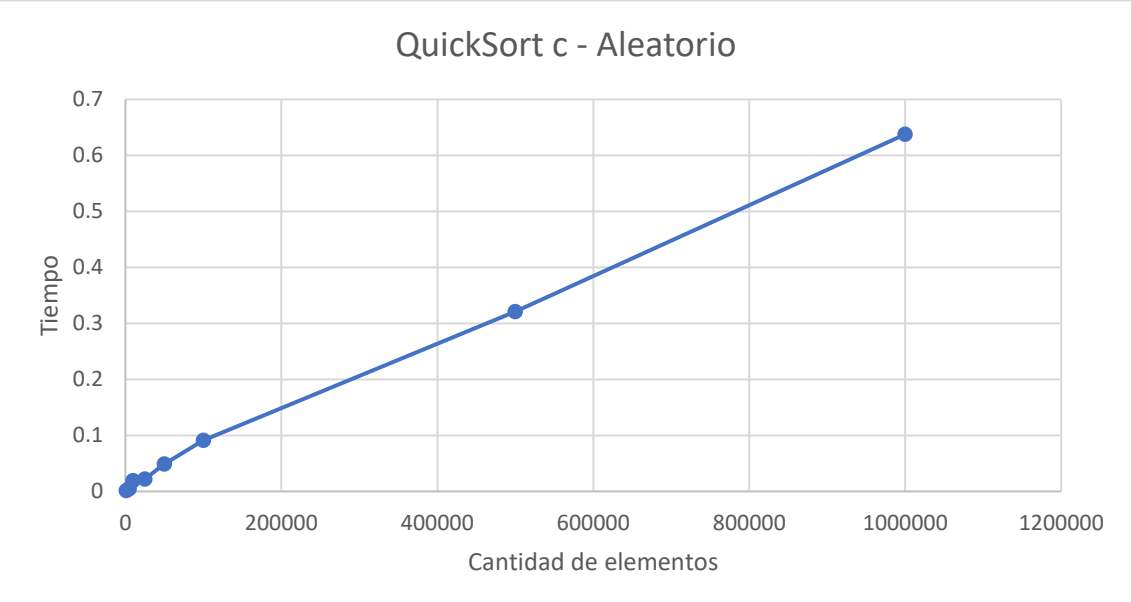
c. Para QuickSort: Una versión que utilice como pivote la media de tres (3) elementos obtenidos de forma aleatoria.

	Prueba No.1 -Aleatoria			
QuickSort c				
Num. Muestra	Num. Elementos	Tiempo de ejecución	Swap elements	Comparation
1	1000	0.001	6959	11336
2	5000	0.004	43092	68682
3	10000	0.03	98223	160996
4	25000	0.022	264073	443312
5	50000	0.037	513074	884557
6	100000	0.086	1044154	1927341
7	500000	0.314	5983195	11432941
8	1000000	0.637	11991337	25720476

	Prueba No.2 -Aleatoria			
QuickSort c				
Num. Muestra	Num. Elementos	Tiempo de ejecución	Swap elements	Comparation
1	1000	0.001	8778	12853
2	5000	0.004	37943	71796
3	10000	0.015	83207	160822
4	25000	0.024	257718	430552
5	50000	0.064	544853	903660
6	100000	0.097	1223458	1992241
7	500000	0.328	5864922	12247453
8	1000000	0.641	12149021	27245881

Prueba No.3 -Aleatoria				
QuickSort c				
Num. Muestra	Num. Elementos	Tiempo de ejecución	Swap elements	Comparation
1	1000	0.002	6950	11525
2	5000	0.007	39609	70649
3	10000	0.013	97294	158590
4	25000	0.021	240310	451368
5	50000	0.046	553718	943160
6	100000	0.091	1154398	1965965
7	500000	0.321	5728195	12971371
8	1000000	0.635	11767110	31120525

Promedio -Aleatoria				
QuickSort c				
Num. Muestra	Num. Elementos	Tiempo de ejecución	Swap elements	Comparation
1	1000	0.00133333	7562.33333	11904.6667
2	5000	0.005	40214.6667	70375.6667
3	10000	0.01933333	92908	160136
4	25000	0.02233333	254033.667	441744
5	50000	0.049	537215	910459
6	100000	0.09133333	1140670	1961849
7	500000	0.321	5858770.67	12217255
8	1000000	0.63766667	11969156	28028960.7

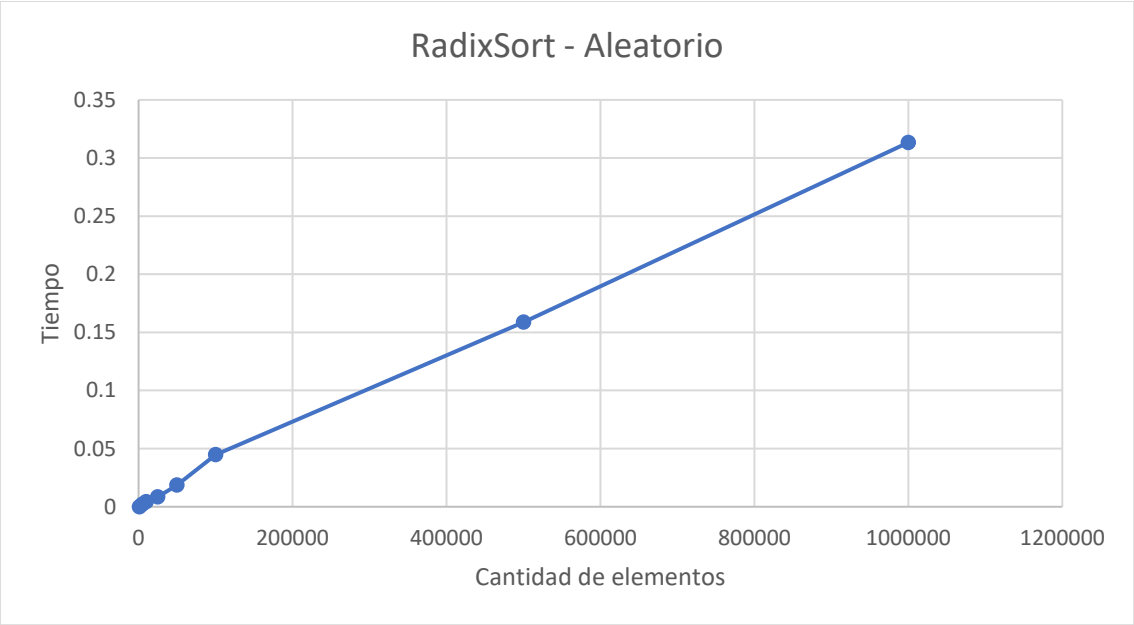


Prueba No.1 -Aleatoria				
RadixSort				
Num. Muestra	Num. Elementos	Tiempo de ejecución	Swap elements	Comparation
1	1000	0	7995	3992
2	5000	0.002	39996	19992
3	10000	0.004	99994	49990
4	25000	0.008	249995	124990
5	50000	0.016	499995	249990
6	100000	0.046	1199993	599988
7	500000	0.164	5999994	2999988
8	1000000	0.32	13999992	6999986

Prueba No.2 -Aleatoria				
RadixSort				
Num. Muestra	Num. Elementos	Tiempo de ejecución	Swap elements	Comparation
1	1000	0	7994	3992
2	5000	0.002	39995	19992
3	10000	0.005	99994	49990
4	25000	0.007	249994	124990
5	50000	0.02	499994	249990
6	100000	0.038	1199993	599988
7	500000	0.156	5999994	2999988
8	1000000	0.31	13999991	6999986

Prueba No.3 -Aleatoria				
RadixSort				
Num. Muestra	Num. Elementos	Tiempo de ejecución	Swap elements	Comparation
1	1000	0	7995	3992
2	5000	0.002	39995	19992
3	10000	0.004	99994	49990
4	25000	0.01	249994	124990
5	50000	0.02	499994	249990
6	100000	0.05	1199993	599988
7	500000	0.156	5999994	2999988
8	1000000	0.31	13999992	6999986

Promedio -Aleatoria				
RadixSort				
Num. Muestra	Num. Elementos	Tiempo de ejecución	Swap elements	Comparation
1	1000	0	7994.66667	3992
2	5000	0.002	39995.3333	19992
3	10000	0.00433333	99994	49990
4	25000	0.00833333	249994.333	124990
5	50000	0.01866667	499994.333	249990
6	100000	0.04466667	1199993	599988
7	500000	0.15866667	5999994	2999988
8	1000000	0.31333333	13999991.7	6999986

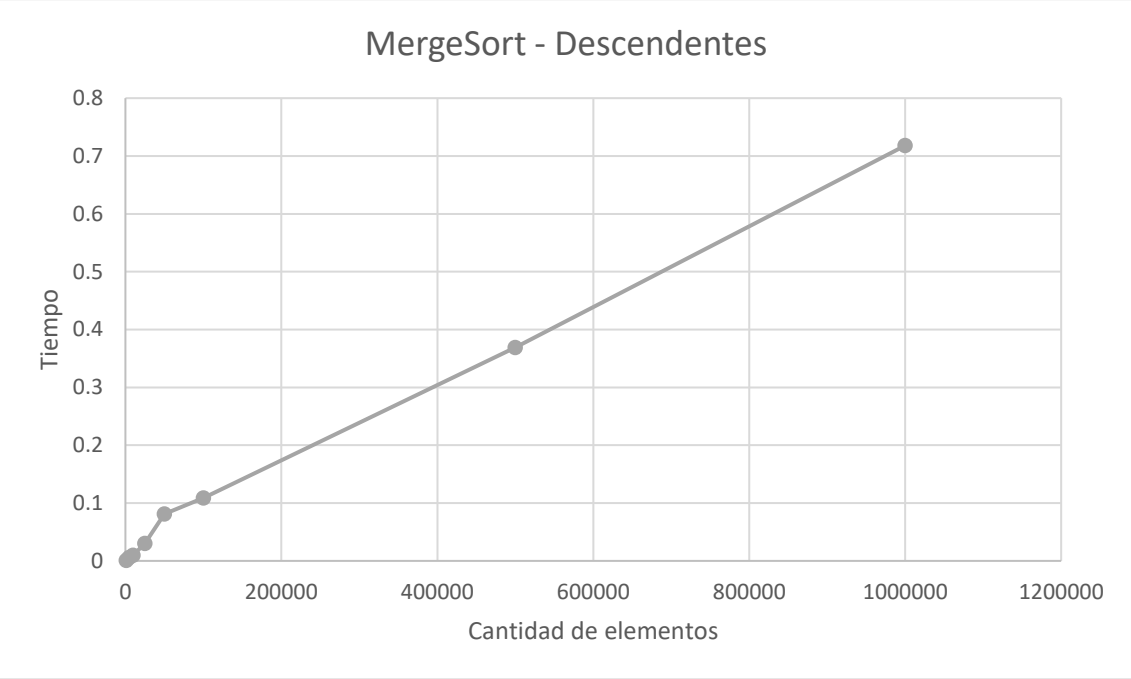


Tiempos de ejecución en casos extremos o peores.

Para este se creó un arreglo con los elementos del mismo de forma descendente para que los algoritmos los organizara de forma ascendente deshabilitando la función de `permutation` y modificando el for que genera los elementos del arreglo y creando una variable nueva llamada baja.

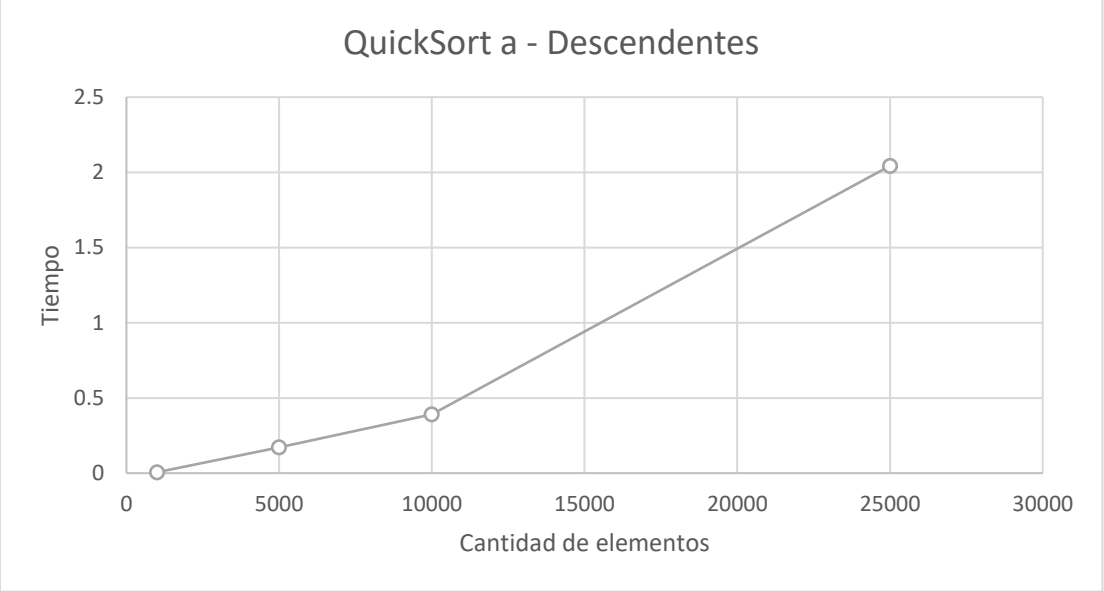
```
/*for (i = 0; i < n; i++)*/for(i = n, baja = 0; i > 0; i--) {
    //vector[i] = i + 1;
    vector[baja] = i - 1;
    baja++;
}
//permutation(vector,n);
```

	Peor de los casos			
Merge Sort				
Num. Muestra	Num. Elementos	Tiempo de ejecución	Swap elements	Comparation
1	1000	0.001	4932	4932
2	5000	0.006	29804	29804
3	10000	0.01	64608	64608
4	25000	0.03	178756	178756
5	50000	0.081	382512	382512
6	100000	0.109	815024	815024
7	500000	0.369	4692496	4692496
8	1000000	0.718	4692496	4692496



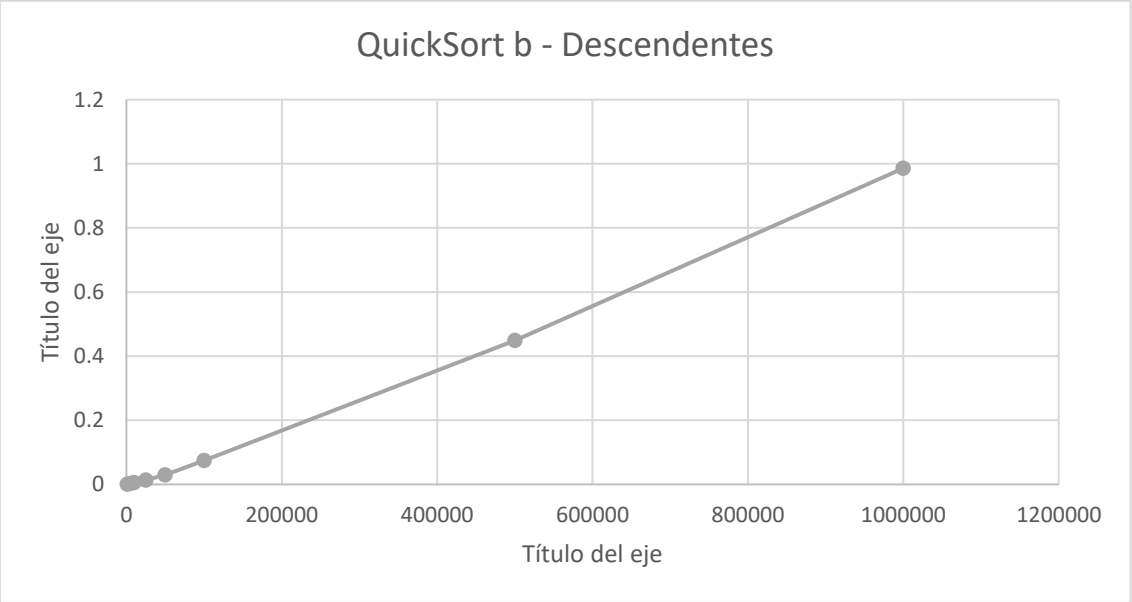
a. Para QuickSort: Una versión que siempre utilice el último elemento del arreglo como el pivote.

	Peor de los casos			
QuickSort				
Num. Muestra	Num. Elementos	Tiempo de ejecución	Swap elements	Comparation
1	1000	0.006	250499	499500
2	5000	0.171	6252499	12497500
3	10000	0.39	25004999	49995000
4	25000	2.043	156262499	312487500
5	50000			
6	100000			
7	500000			
8	1000000			



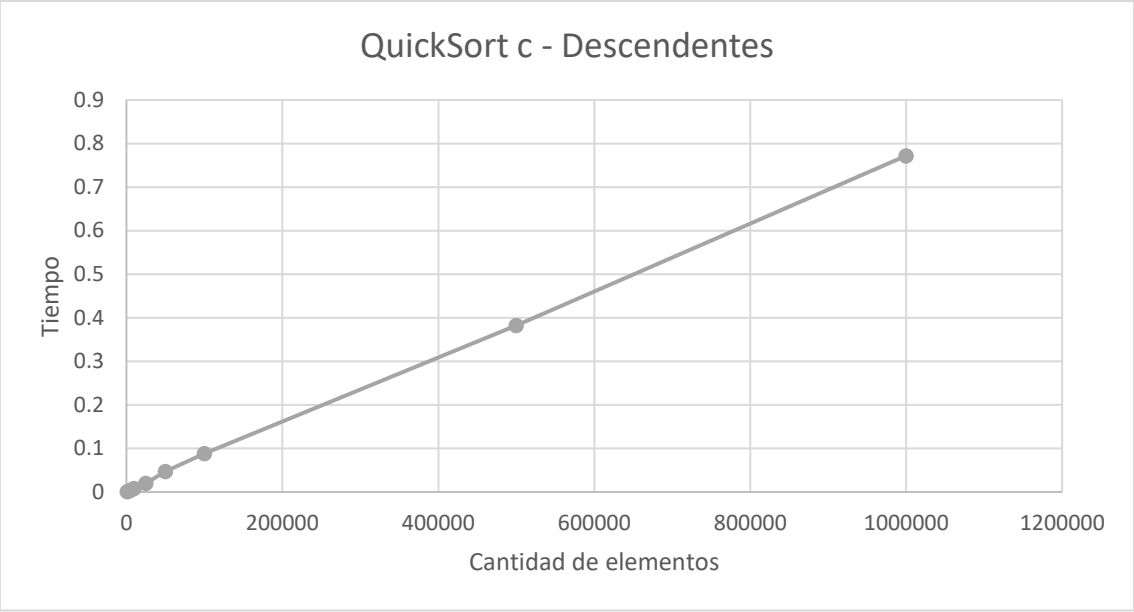
b. Para QuickSort: Una versión que utilice como pivote un elemento aleatorio del mismo.

	Peor de los casos			
QuickSort				
Num. Muestra	Num. Elementos	Tiempo de ejecución	Swap elements	Comparation
1	1000	0.001	6486	11109
2	5000	0.003	42973	70996
3	10000	0.006	90359	155323
4	25000	0.014	254901	431052
5	50000	0.03	509584	894073
6	100000	0.075	1120767	1925613
7	500000	0.449	12558017	16515136
8	1000000	0.986	39143611	47097700

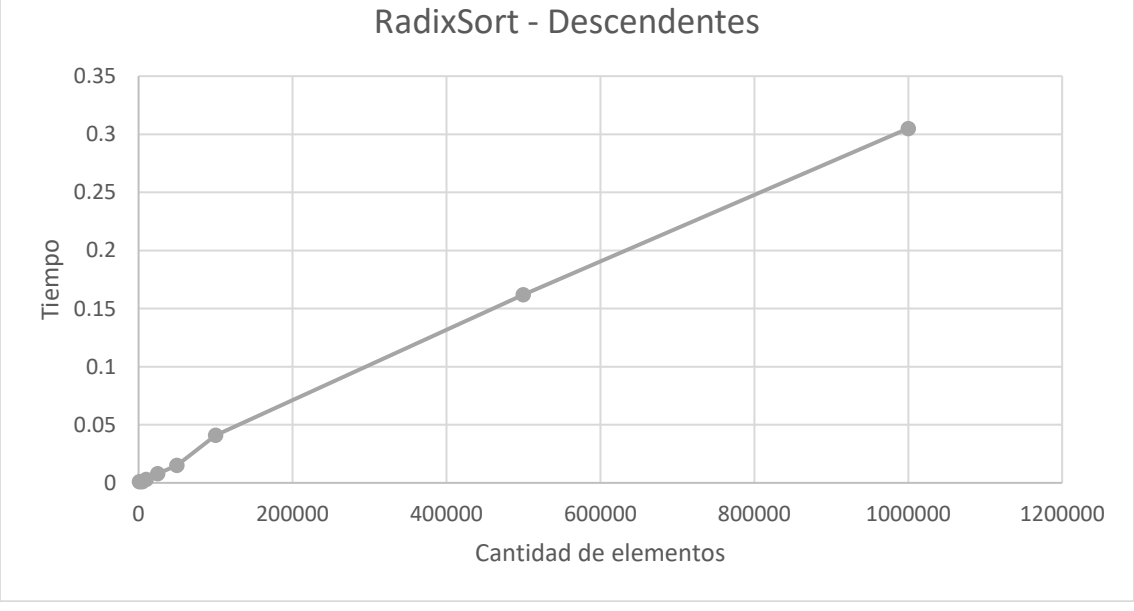


c. Para QuickSort: Una versión que utilice como pivote la media de tres (3) elementos obtenidos de forma aleatoria.
Si existen menos de tres elementos, usted deberá decidir la estrategia del pivote a utilizar.

	Peor de los casos			
QuickSort				
Num. Muestra	Num. Elementos	Tiempo de ejecución	Swap elements	Comparation
1	1000	0.001	6146	10291
2	5000	0.004	41300	67670
3	10000	0.008	93007	149328
4	25000	0.02	262758	421123
5	50000	0.047	552069	886629
6	100000	0.088	1294811	1995184
7	500000	0.382	14644044	18315196
8	1000000	0.772	33880179	41585660



	Peor de los casos			
RadixSort				
Num. Muestra	Num. Elementos	Tiempo de ejecución	Swap elements	Comparation
1	1000	0.001	5994	2994
2	5000	0.001	39992	19992
3	10000	0.003	79992	39992
4	25000	0.008	249991	124990
5	50000	0.015	499990	249990
6	100000	0.041	999990	499990
7	500000	0.162	5999988	2999988
8	1000000	0.305	11999988	5999988



Análisis de resultados.

Comparación de Resultados Teóricos y

Prácticos

Para poder observar el comportamiento expuesto de forma teórica por el maestro, se propuso y se realizó la práctica de graficar los datos obtenidos para mostrar el crecimiento del tiempo proporcional al número de elementos que se encuentra bajo el estudio de los algoritmos de ordenamientos, Se puedo ver esta relación con las gráficas mostradas anteriormente en el procedimiento.

Algoritmo de ordenamiento Merge Sort

	Complejidad		
	Mejor de los casos	Caso intermedio	Peor de los casos
	O(n*log n)	O(n*log n)	O(n*log n)
Merge Sort		Intercambios (Swaps)	
	O(n)	O(n)	O(n)

Siendo el algoritmo de ordenamiento recursivo Merge Sort siendo un algoritmo de ordenamiento estable lo que significa que la cantidad de elementos están ordenados en la misma forma que el orden de una lista de elementos ya ordenados de forma ascendente. Observando los datos obtenidos del algoritmo de ordenamiento Merge Sort durante el procedimiento de realización de cada complejidad temporal se puede observar cómo lo teórico correspondió a los resultados prácticos ya que se puede ver en los tres caos de complejidad temporal, como su grafico corresponde al inicio a una pequeña curva de la función O (n*log n) siendo esta misma constante o parecida en los 3 casos. Incluso en el intermedio que hasta su promedio resulto siendo esta grafica demostrando que su definición teórica corresponde a su aplicación práctica.
También observando como en el caso promedio con 100000 elementos el tiempo que tomo fue de 0.124666667 y su con su siguiente caso que es 500000 tiene 5 veces más sus elementos su tiempo tendría que ser de t = 0.124666667 (5*log 5) cuyo resultado es 0.43569 teniendo de margen de error de un 24.40% al resultado práctico de 0.44467 demostrando como es consecuente uno con otro, resultando un numero cercano.

a. Para QuickSort: Una versión que siempre utilice el último elemento del arreglo como el pivote.

	Complejidad		
	Mejor de los casos	Caso intermedio	Peor de los casos
QuickSort a	O(n2)	O(n*log n)	O(n2)
		Intercambios (Swaps)	
	O(n)	O(n)	O(n)

Se puede observar cómo lo teórico corresponde con lo practico dando correspondencia la grafica con sus fórmulas, se puede observar claramente esta relación al observar por ejemplo la grafica del comportamiento cuando los elementos están desorganizados de forma aleatoria o "Caso intermedio", en donde el tiempo por ejemplo en los 100000 elementos es de 0.034 y al aplicar la formula 0.034(5*log 5) siendo 0.119 esta cercano al siguiente tiempo de 0.135 con un margen de error de 35.94%.

.

b. Para QuickSort: Una versión que utilice como pivote un elemento aleatorio del mismo.

	Complejidad		
	Mejor de los casos	Caso intermedio	Peor de los casos
QuickSort b	O (n*log n)	O (n*log n)	O (n*log n)
		Intercambios (Swaps)	
	O(n)	O(n)	O(n)

Observando las gráficas y resultados obtenidos tras la corrida del programa se puede decir que lo teórico claramente corresponde a lo practico y observando la claridad de esto en el "Peor de los casos" en donde los elementos están arreglados de forma descendentes. Por ejemplo, en el tiempo de 100000 elementos en el caso intermedio es de 0.07133333 y aplicando la formula 0.07133333(5*log5) el resultado es de 0.249 siendo un aproximado del siguiente de 0.273 con un margen de error de 33.53%.

c. Para QuickSort: Una versión que utilice como pivote la media de tres (3) elementos obtenidos de forma aleatoria.

	Complejidad		
	Mejor de los casos	Caso intermedio	Peor de los casos
	O (n)	O(n*log n)	O (n)
QuickSort c		Intercambios (Swaps)	
	O(n)	O(n)	O(n)

Se puede observar con los resultados como concuerdan con lo expuesto de forma teórica en donde la media de 3 elementos obtenidos de forma aleatoria es la mejor forma de resolver el ordenamiento de algoritmos dentro de las formas de QuickSort, en donde se observa como O(n) es el mejor de los casos para este tipo de forma de este algoritmo de ordenamiento esto se observa por ejemplo en los 100000 elementos del caso intermedio en donde su tiempo es de 0.091333333 que aplicando la fórmula 0.091333333(5*log5) es de 0.319 siendo aproximado al siguiente que es de 0.321 con un margen de error de 9.37% siendo el menor margen de error de las 3 formas.

Algoritmo de ordenamiento Radix Sort

	Complejidad		
	Mejor de los casos	Caso intermedio	Peor de los casos
	O(kn)	O(kn)	O(kn)
Radix Sort		Intercambios (Swaps)	
	O(n)	O(n)	O(n)

Como se sabe que el Radix Sort tiene una complejidad de tiempo lineal que es mejor que O (nlog n) de los algoritmos de ordenamiento comparativo. Al observar los datos que se lograron obtener se puede observar cómo los mismo se encuentran apegados a la teoría, ya que estos ya tabulados y graficados fueron los que se esperaban obtener tras su prueba de corrida. Observando como dice la teoría la complejidad del tipo de radix es O (kn) para n claves que son números enteros de tamaño de palabra k. Para todos los casos, el tiempo, es decir, el mejor, el peor y la complejidad de tiempo promedio es O (kn). Este comportamiento se observa en las gráficas del Radix siendo estas una proporcionalidad lineal, observando como esta es una línea recta demostrando que la teoría no es solamente conceptual sino también práctica.

Mejor de los casos, peor de los casos y caso intermedio.

Se puede observar cómo cada uno de los algoritmos cumple con su definición teórica siendo por ejemplo en el caso de MergeSort sus 3 casos el mejor, el peor y el intermedio O(n*logn) como se puede observar en las gráficas, aunque no muy detallado pero su aplicación práctica cumplida de cierta forma. En el caso del QuickSort en donde el pivote es el último elemento de los resultados obtenidos se puede inferir que el "Mejor de los casos" en donde los números ya están organizados y "Peor de los casos" en donde los números están organizados de forma descendentes terminaron siendo ambos los peores casos O(n2) por los que podría pasar el QuickSort en donde utiliza el último elemento del arreglo como el pivote en donde el mejor de los casos se dio en el "Caso intermedio" donde el arreglo tenía sus elementos desordenados aleatoriamente O(n*logn), demostrando que lo dicho anterior mente con lo teórico tiene fuerte relación al comportamiento practico del mismo, además siendo el caso en donde el pivote es la media de 3 elementos aleatorios el mejor de los casos para la forma de ordenamiento del algoritmo. En el caso del RadixSort se comprobó que también su descripción teórica corresponde a su aplicación grafica en donde su complejidad para el mejor, peor y caso intermedio siempre es igual a O(kn) como se ve tanto en los resultados numéricos como los gráficos.

Numero de comparaciones e intercambios

En los números de comparación e intercambios se observar como el QuickSort fue el que tuvo el mayor número de estos tanto como en sus 3 formas como en dos 2 de sus casos siendo estos el "Peor" donde se encuentran los elementos de forma descendentes y el mejor donde se encuentra los elementos de forma ascendentes, en comparación del RadixSort y MergeSort que solo fue mayor en tema de sus intercambios en el "caso intermedio" en donde los elementos se encontraban de forma aleatoria desorganizados, debido a que el QuickSort es un algoritmo que se conoce por implementar la función de que el numero sea comparado por el pivote siendo mayor que o menor que este repitiéndose e intercambiando tantas maneras se requiera para organizar la lista de elementos, a diferencia en casos como el de Mergesort en donde se unen dos estructuras ordenadas para formar una sola ordenada de forma correcta, además que el mismo utiliza arreglos auxiliares para intercambiarlos. Y en el caso del Radixsort se empieza con el dígito menos significativo y se acomoda la lista de elementos en base en el dígito de esa posición, luego se cambia al siguiente dígito y se vuelve a acomodar la lista en base a ese dígito y así consecutivamente hasta hacerse con todos los dígitos.

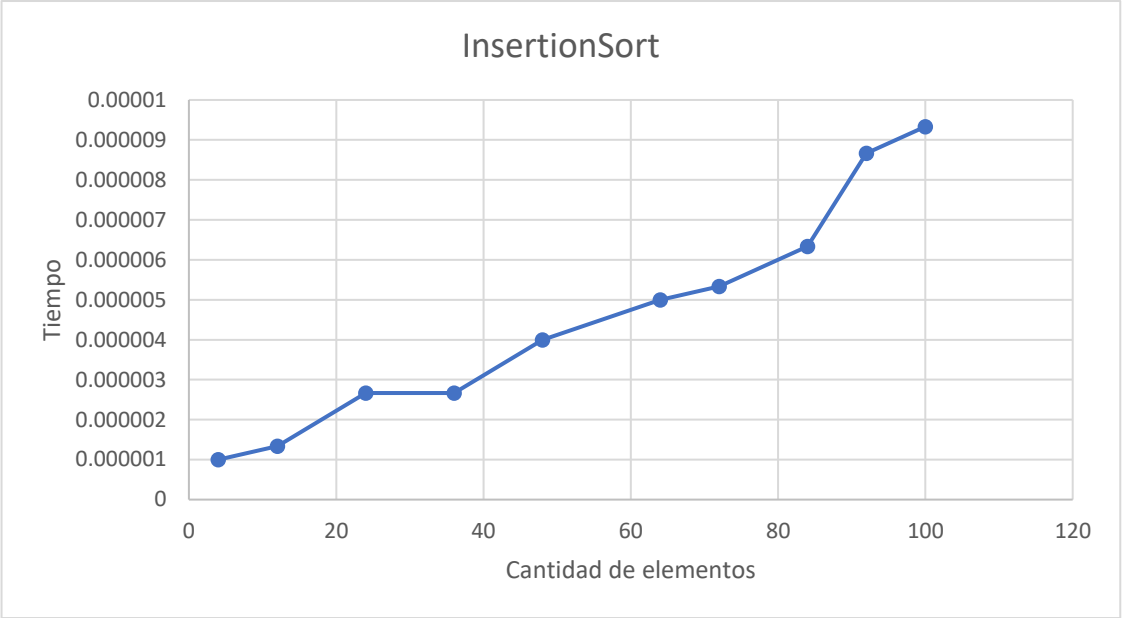
Corridas cortas

Prueba No.1 -Aleatoria				
InsertionSort				
Num. Muestra	Num. Elementos	Tiempo de ejecución	Swap elements	Comparation
1	4	0.000001	1	4
2	12	0.000002	32	43
3	24	0.000002	124	147
4	36	0.000002	337	372
5	48	0.000004	503	550
6	64	0.000005	982	1045
7	72	0.000005	1099	1170
8	84	0.000007	1697	1780
9	92	0.000011	2042	2133
10	100	0.000012	2236	2335

Prueba No.2 -Aleatoria				
InsertionSort				
Num. Muestra	Num. Elementos	Tiempo de ejecución	Swap elements	Comparation
1	4	0.000001	5	8
2	12	0.000001	24	35
3	24	0.000002	111	134
4	36	0.000003	277	312
5	48	0.000003	520	567
6	64	0.000005	1037	1100
7	72	0.000006	1375	1446
8	84	0.000006	1555	1638
9	92	0.000008	2363	2454
10	100	0.000008	2666	2765

Prueba No.3 -Aleatoria				
InsertionSort				
Num. Muestra	Num. Elementos	Tiempo de ejecución	Swap elements	Comparation
1	4	0.000001	4	7
2	12	0.000001	39	50
3	24	0.000004	162	185
4	36	0.000003	334	369
5	48	0.000005	595	642
6	64	0.000005	1009	1072
7	72	0.000005	1265	1336
8	84	0.000006	1787	1870
9	92	0.000007	2002	2093
10	100	0.000008	2523	2622

Promedio -Aleatoria				
InsertionSort				
Num. Muestra	Num. Elementos	Tiempo de ejecución	Swap elements	Comparation
1	4	0.000001	3.333333333	6.333333333
2	12	1.33333E-06	31.66666667	42.66666667
3	24	2.66667E-06	132.3333333	155.3333333
4	36	2.66667E-06	316	351
5	48	0.000004	539.3333333	586.3333333
6	64	0.000005	1009.333333	1072.333333
7	72	5.33333E-06	1246.333333	1317.333333
8	84	6.33333E-06	1679.666667	1762.666667
9	92	8.66667E-06	2135.666667	2226.666667
10	100	9.33333E-06	2475	2574

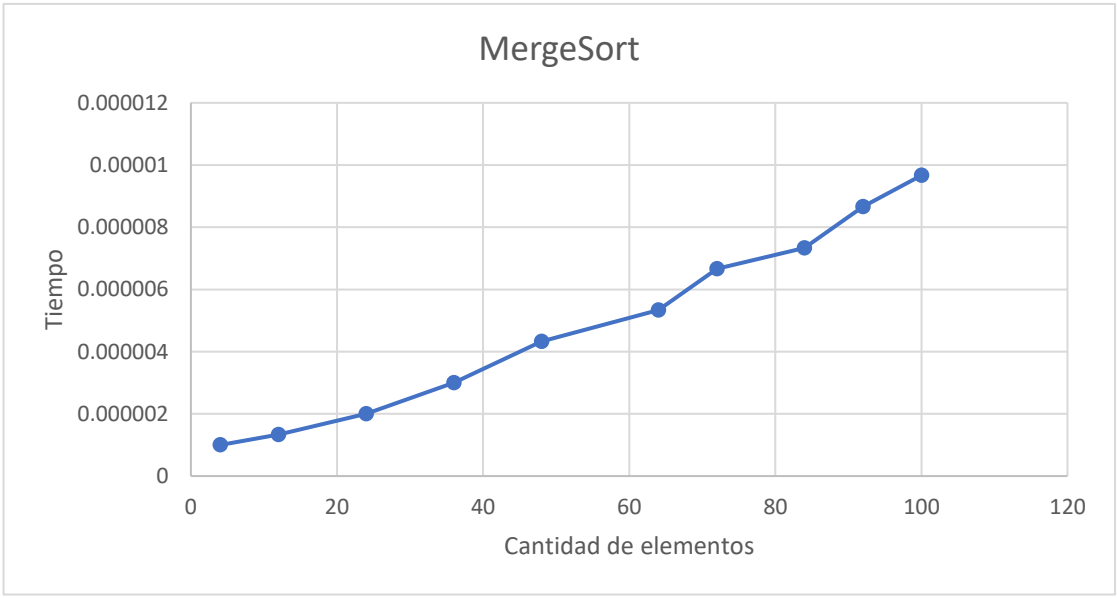


Prueba No.1 -Aleatoria				
MergeSort				
Num. Muestra	Num. Elementos	Tiempo de ejecución	Swap elements	Comparation
1	4	0.000001	5	5
2	12	0.000002	28	28
3	24	0.000002	84	84
4	36	0.000003	144	144
5	48	0.000005	210	210
6	64	0.000006	300	300
7	72	0.000007	362	362
8	84	0.000009	430	430
9	92	0.00001	488	488
10	100	0.000011	542	542

Prueba No.2 -Aleatoria				
MergeSort				
Num. Muestra	Num. Elementos	Tiempo de ejecución	Swap elements	Comparation
1	4	0.000001	5	5
2	12	0.000001	31	31
3	24	0.000002	83	83
4	36	0.000003	140	140
5	48	0.000004	214	214
6	64	0.000006	308	308
7	72	0.000006	359	359
8	84	0.000006	429	429
9	92	0.000008	491	491
10	100	0.000009	536	536

Prueba No.3 -Aleatoria				
MergeSort				
Num. Muestra	Num. Elementos	Tiempo de ejecución	Swap elements	Comparation
1	4	0.000001	5	5
2	12	0.000001	26	26
3	24	0.000002	78	78
4	36	0.000003	142	142
5	48	0.000004	215	215
6	64	0.000004	303	303
7	72	0.000007	354	354
8	84	0.000007	436	436
9	92	0.000008	489	489
10	100	0.000009	545	545

Promedio -Aleatoria				
MergeSort				
Num. Muestra	Num. Elementos	Tiempo de ejecución	Swap elements	Comparation
1	4	0.000001	5	5
2	12	1.33333E-06	28.33333333	28.33333333
3	24	0.000002	81.66666667	81.66666667
4	36	0.000003	142	142
5	48	4.33333E-06	213	213
6	64	5.33333E-06	303.6666667	303.6666667
7	72	6.66667E-06	358.3333333	358.3333333
8	84	7.33333E-06	431.6666667	431.6666667
9	92	8.66667E-06	489.3333333	489.3333333
10	100	9.66667E-06	541	541

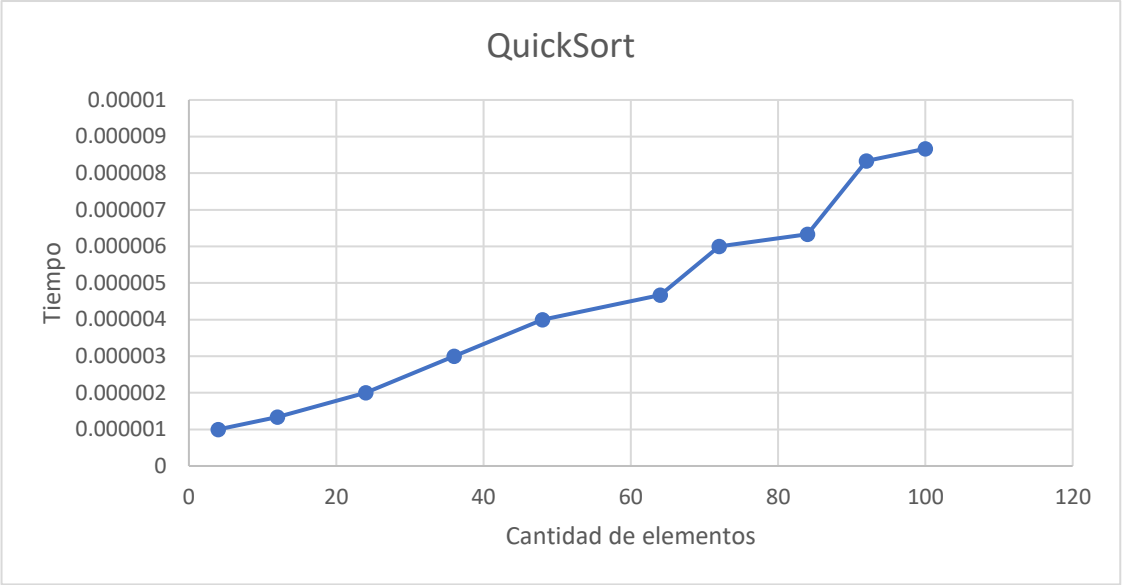


Prueba No.1 -Aleatoria				
QuickSort				
Num. Muestra	Num. Elementos	Tiempo de ejecución	Swap elements	Comparation
1	4	0.000001	7	4
2	12	0.000001	23	32
3	24	0.000002	68	76
4	36	0.000003	167	208
5	48	0.000004	167	210
6	64	0.000005	242	329
7	72	0.000006	264	372
8	84	0.000006	340	458
9	92	0.000009	421	521
10	100	0.00001	397	658

Prueba No.2 -Aleatoria				
QuickSort				
Num. Muestra	Num. Elementos	Tiempo de ejecución	Swap elements	Comparation
1	4	0.000001	6	6
2	12	0.000001	27	38
3	24	0.000002	77	85
4	36	0.000003	122	155
5	48	0.000004	169	230
6	64	0.000005	242	344
7	72	0.000007	274	397
8	84	0.000007	322	545
9	92	0.000009	385	507
10	100	0.000009	553	836

Prueba No.3 -Aleatoria				
QuickSort				
Num. Muestra	Num. Elementos	Tiempo de ejecución	Swap elements	Comparation
1	4	0.000001	7	4
2	12	0.000002	25	35
3	24	0.000002	74	79
4	36	0.000003	105	152
5	48	0.000004	156	225
6	64	0.000004	332	375
7	72	0.000005	271	400
8	84	0.000006	378	504
9	92	0.000007	459	571
10	100	0.000007	454	608

Promedio -Aleatoria				
QuickSort				
Num. Muestra	Num. Elementos	Tiempo de ejecución	Swap elements	Comparation
1	4	0.000001	6.666666667	4.666666667
2	12	1.33333E-06	25	35
3	24	0.000002	73	80
4	36	0.000003	131.3333333	171.6666667
5	48	0.000004	164	221.6666667
6	64	4.66667E-06	272	349.3333333
7	72	0.000006	269.6666667	389.6666667
8	84	6.33333E-06	346.6666667	502.3333333
9	92	8.33333E-06	421.6666667	533
10	100	8.66667E-06	468	700.6666667



Como se puede ver para estos casos de corridas cortas el MergeSort tiene resultados un poco mas exactos o “mejores” a comparación del InsertionSort y QuickSort esto se puede observar en la grafica del mismo ya que es mas cercana a la de resultados anteriores en el tema de intercambios y comparaciones concuerdan con los resultados obtenidos anteriormente y se puede notar como en el InsertionSort y QuickSort los valores son opuestos es decir si por ejemplo en Insertion dio 4 intercambios y 7 comparaciones en el QuickSort pasa todo lo contrario en donde 7 es el numero de intercambios y 4 de comparaciones. Ademas comparando con los resultados anteriores al parecer ya que la cantidad de elementos es mucho menor esto resula a que la toma de tiempo del programa muestre resultados algo “raros”. Cabe destacar que a diferencia del método Quicksort que divide la estructura en dos y ordena cada mitad recursivamente. El caso del MergeSort es el opuesto, es decir, en este método se unen dos estructuras ordenadas para formar una sola ordenada correctamente. Y el InsertionSort Consta de tomar uno por uno los elementos de un arreglo y recorrerlo hacia su posición con respecto a los anteriormente ordenados.

Teoría y Conceptos de Algoritmos de Ordenamiento

1. Estabilidad de los algoritmos de ordenamiento.

   Por estabilidad se puede referir a que un ordenamiento estable mantiene el orden relativo que tenían originalmente los elementos con claves iguales. Por ejemplo, si una lista ordenada por fecha se reordena en orden alfabético con un algoritmo estable, todos los elementos cuya clave alfabética sea la misma quedarán en orden de fecha. Como lo es MergeSort por ejemplo.

2. Algoritmos de Ordenamiento Híbridos.

   Estos tipos de algoritmos son muy comunes en diversas implementaciones del mundo real para así optimizar algoritmos recursivos, en particular implementaciones que usan técnicas como divide y vencerás, donde el tamaño de los datos decrece a medida que la profundidad de la recursión aumenta. Como es el caso del QuickSort.

3. Límite inferior al mejor caso para los algoritmos basados en comparación de claves.

   El algoritmo es óptimo, si orden del límite inferior es igual que el orden del límite superior

Conclusión

Con la realización de esta práctica se logró cumplir con el objetivo planteado de la misma, a través de los elementos especificados en el ambiente de trabajo en donde se ejecutaron y se tomó el tiempo de los algoritmos de ordenamiento, además del análisis de los resultados logrando comparar los resultados prácticos con los resultados teóricos siendo los mismos los esperados con la realización de esta.

Se pudo afirmar los conceptos teóricos de los algoritmos de ordenamiento MergeSort, QuickSort y RadixSort, a través de su aplicación práctica y toma de tiempo donde se distingue que el MergeSort es un algoritmo de ordenamiento estable, ya su fórmula para todos sus casos siempre será O(n*logn) como se muestra en sus tabulaciones y la curva representada en sus gráficas, a diferencia de sus compañeros algorítmicos QuickSort, en donde dependiendo la forma de obtener el pivote afecta en la obtención de los valores y graficas de sus casos y RadixSort, en donde su complejidad siempre será O(kn) o O(n), en donde estos algoritmos demostraron que el mejor de los caos para los dos siempre será confirmando lo propuesto por el docente donde el mismo relativa que “el mejor de los casos entre estos algoritmos será O(nlogn) aunque a veces, pero también como en la Practica 1-A se verá O(n) o O(kn) como mejor de sus casos y que se notara en la duración de los algoritmos, pero OJO! esto no aplica de igual forma para los 3” confirmando lo dicho a través de las gráficas y tomas de tiempo.