I Tarea Programada

Berta Sánchez Jalet B66605

1 Introducción

En el siguiente trabajo se analizan los algoritmos de ordenamiento Selección, Inserción, Mezcla, Heap Sort, Quick Sort y Radix Sort. Como problema adicional, se encontrará la mediana de un arreglo, por medio del algoritmo Partition, el cual es utilizado para la resolución de Quick Sort. También se expondrá en una tabla los tiempos promedio de cada algoritmo para cuatro tamaños diferentes. Sus gráficas luego serán analizadas y comparadas con sus respectivos tiempos de duración.

2 Metodología

Los tres diferentes algoritmos de ordenadmiento fueron probados una serie de cinco veces y luego fue sacado su promedio. Esto se realizó para cuatro tamaños diferentes, 100000, 200000, 300000 y 400000. Se utilizó la libreria de C++ llamada chrono para medir en segundos la duración de cada algoritmo.

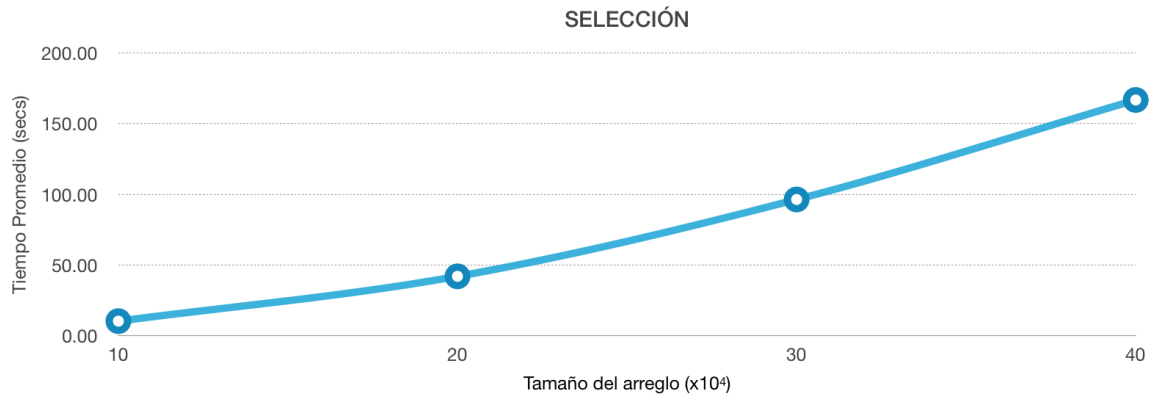
Tiempo de Ejecución de los Algoritmos

		Tiempo (secs)					
		Corrida					
Algoritmos	**Tamaño (x10^4)**	**1**	**2**	**3**	**4**	**5**	**Promedio**
Selección	10	10.44	10.37	10.44	10.09	10.11	10.29
	20	41.88	39.96	41.61	43.21	43.35	42.00
	30	97.00	96.41	96.98	95.92	95.60	96.38
	40	168.03	168.37	168.70	164.59	164.25	166.78
Inserción	10	6.56	6.39	6.57	6.20	6.18	6.38
	20	24.20	24.00	25.73	26.89	26.29	25.42
	30	59.82	59.55	59.84	58.66	58.95	59.36
	40	102.90	104.55	104.79	101.15	101.35	102.94
Mezcla	10	0.018	0.016	0.016	0.015	0.019	0.016
	20	0.034	0.032	0.037	0.033	0.042	0.035
	30	0.060	0.052	0.058	0.055	0.050	0.055
	40	0.078	0.071	0.075	0.071	0.074	0.073
Heap Sort	10	0.039	0.038	0.038	0.039	0.039	0.038
	20	0.082	0.080	0.081	0.082	0.082	0.081
	30	0.126	0.127	0.138	0.137	0.127	0.131
	40	0.170	0.172	0.170	0.167	0.168	0.169
Quick Sort	10	0.136	0.138	0.136	0.135	0.133	0.135
	20	0.500	0.495	0.494	0.494	0.493	0.495
	30	1.087	1.107	1.107	1.146	1.097	1.108
	40	1.951	1.967	1.938	1.948	1.915	1.943
Radix Sort	10	0.0105	0.0106	0.0106	0.0109	0.0105	0.0106
	20	0.022	0.021	0.022	0.022	0.021	0.021
	30	0.032	0.033	0.034	0.034	0.034	0.033
	40	0.045	0.045	0.045	0.045	0.044	0.044

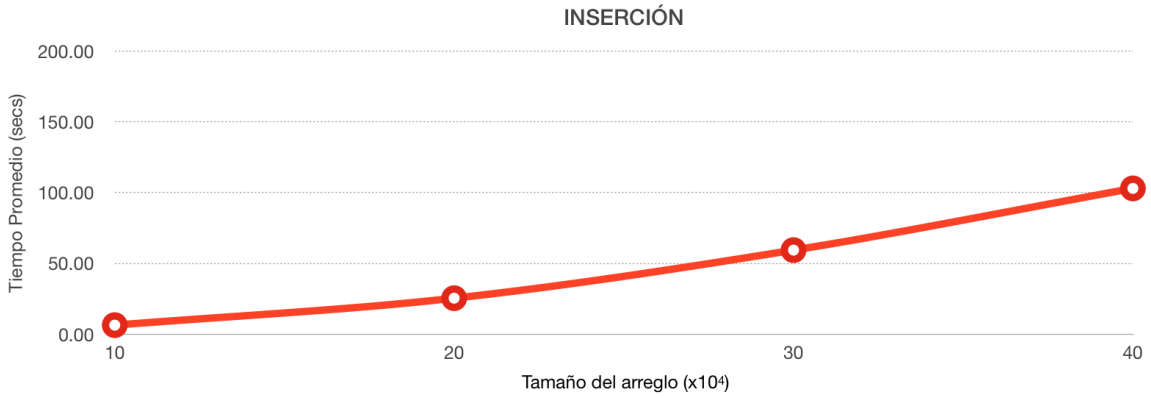
3 Resultados

Como se puede observar en las siguientes gráficas, cada algoritmo crece conforme su tamaño aumenta, sin embargo se puede ver la gran diferencia que hay entre el tiempo de duración de los algoritmos de Selección e Inserción contra el tiempo que tardo Mezcla. Esta diferencia ocurre ya que al estudiar cada algoritmo, nos damos cuenta que la implementación de cada uno hace que sean más eficientes unos que otros. Como lo hemos estudiado en clase, al analizar Selección e Inserción los ordenes de duración de estos algoritmos son $\Theta(n^2)$ mientras que el algoritmo de Mezcla tiene un orden de duracion de $\Theta(n \cdot log\ n)$. Lo anterior se puede ver claramente al gráficar los algoritmos ya que se puede ver como las lineas tienden a sus respectivas funciones como se puede ver el la figura 4.1. Fue una sorpresa al probar Quick Sort y Heap Sort ya que esperaba ver que los resultados de Quick Sort fueran los más rápidos, pero en realidad Heap Sort fue el más rápido de estos dos. Me pareció muy interesante Radix Sort ya que es el único algoritmo que no está basado en comparaciones directas, haciendo que tenga un orden de duración de $\Theta(n)$ y por lo tanto es el más rápido de todos. Todos estos resultados se pueden ver graficados en las siguientes páginas por su cuenta y al final una gráfica mostrando la comparación de todos los algoritmos. Hay que tomar en cuenta que esta última gráfica esta en formato logarítmico y no lineal.

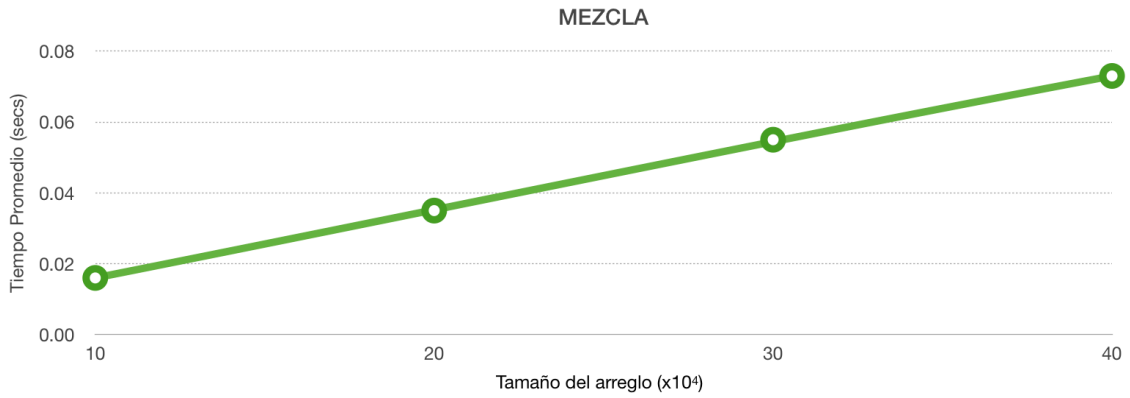
3.1



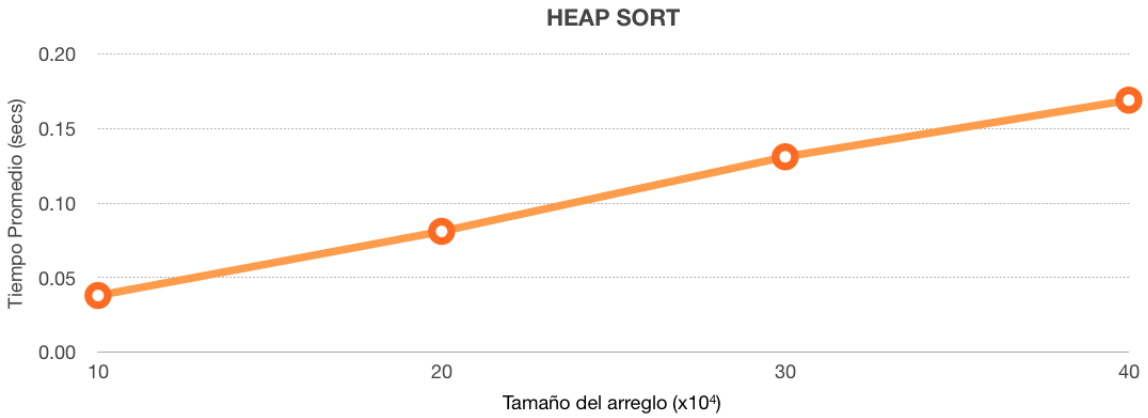
3.2



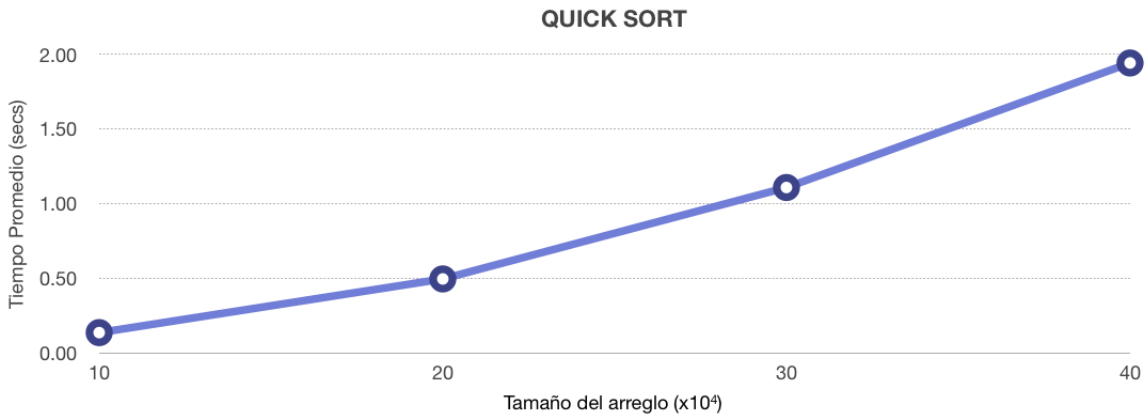
3.3


MEZCLA
0.08
0.06
0.04
0.02
0.00
Tiempo Promedio (secs)
10
20
30
40
Tamaño del arreglo (x10⁴)

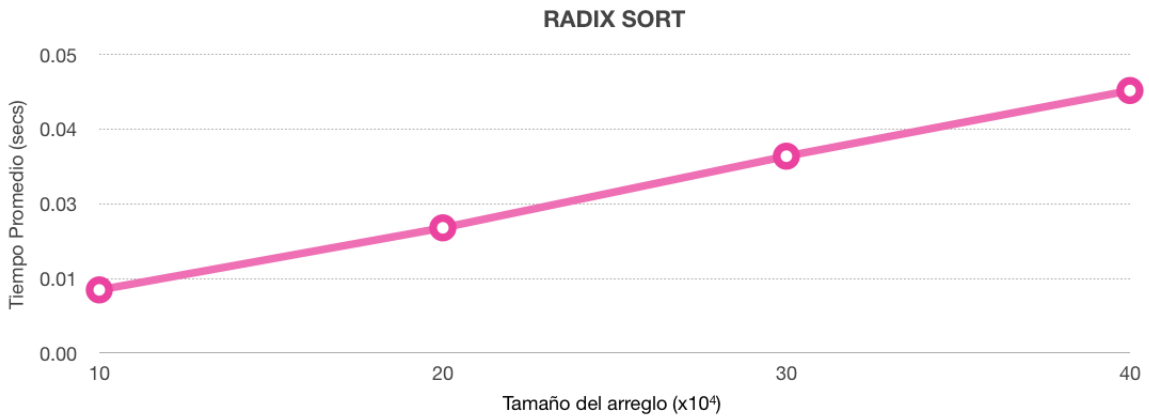
3.4


HEAP SORT
0.20
0.15
0.10
0.05
0.00
Tiempo Promedio (secs)
10
20
30
40
Tamaño del arreglo (x10⁴)

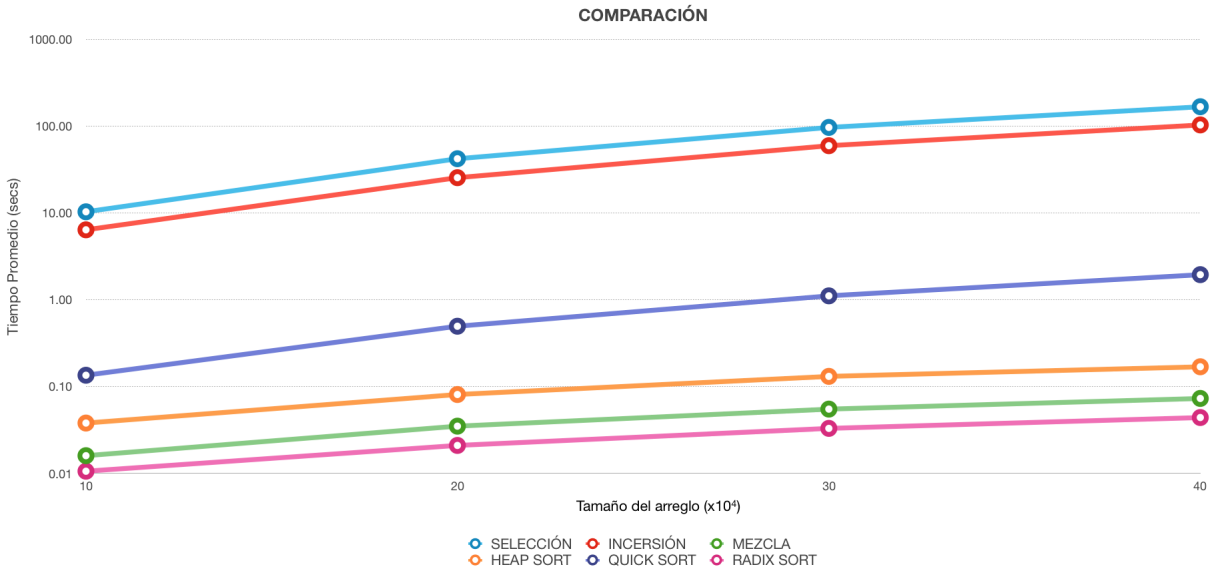
3.5


QUICK SORT
2.00
1.50
1.00
0.50
0.00
Tiempo Promedio (secs)
10
20
30
40
Tamaño del arreglo (x10⁴)

3.6


RADIX SORT
0.05
0.04
0.03
0.01
0.00
Tiempo Promedio (secs)
10
20
30
40
Tamaño del arreglo (x10⁴)

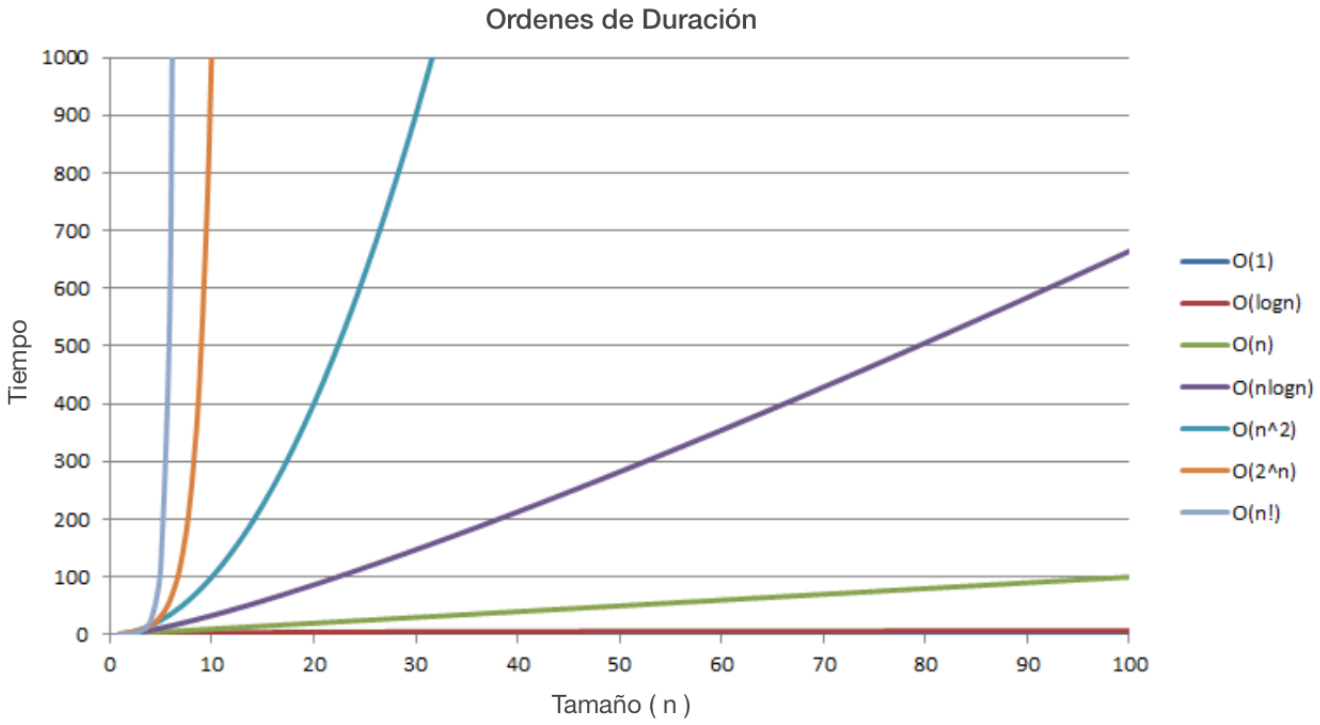
3.7


COMPARACIÓN
1000.00
100.00
10.00
1.00
0.10
0.01
Tiempo Promedio (secs)
10
20
30
40
Tamaño del arreglo (x10⁴)
SELECCIÓN
INCERSIÓN
MEZCLA
HEAP SORT
QUICK SORT
RADIX SORT

4 Conclusiones

Se puede concluir que los resultados obtenidos sí fueron los esperados, ya que se puede ver la representación gráfica de cada algoritmo y la forma en la que este crece. Al colocar las gráficas juntas se puede ver la gran diferencia en tiempo que hay entre los tres algoritmos, lo que nos muestra que la manera en la que uno implementa un algoritmo de verdad afecta su eficiencia a la hora de correr nuestros programas.

4.1



5 Apéndice

5.1 Código de los Algoritmos

Selección (int * arreglo, int tamaño)

```
if (tamaño > 1) {
    int min;
    int aux;
    for ( int i = 0;  i < tamaño;  ++i ) {
        min = i;
        for ( int j = i;  j < tamaño;  ++j ) {
            if (arreglo[ j ] < min) {
            min = j;
            }
        }
    aux = arreglo[ i ];
    arreglo[ i ] = arreglo[min];
    arreglo[min] = aux;
    }
}
```

Inserción (int * arreglo, int tamaño)

```
if (tamaño > 1) {
    int key;
    int i;
    for ( int j = 1;  j < tamaño;  ++j ) {
        key = arreglo[ j ];
        i = j - 1;
        while ( i >= 0 && arreglo[ i ] > key) {
            arreglo[i+1] = arreglo[ i ];
            --i;
        }
    arreglo[i+1] = key;
    }
}
```

Merge Sort (int * arreglo, int l, int r)

```
if (l < r) {
    int m = floor( (l + (r - 1) )/2);
    Merge Sort (arreglo, l, m);
    Merge Sort (arreglo, m + 1, r);
    Merge (arreglo, l, m, r);
}
```

Merge (int * arreglo, int l, int m, int r)

```
int nL = m - l + 1;
int nR = r - m;
int L[nL];
int R[nR];
int i, j, k;

for (i = 0; i < nL; ++i) {
    L[ i ] = arreglo[l+i];
}
for (j = 0; j < nR; ++j) {
    R[ j ] = arreglo[m+1+j];
}

// Se reinician los índices.
i = 0;
j = 0;
k = l;

while (i < nl && j < nR) {
    if (L[ i ] <= R[ j ]) {
        arreglo[ k ] = L[ i ];
        ++i;
    } else {
        arreglo[ k ] = R[ j ];
        ++j;
    }
++k;
}

// Pone los elementos restantes en el arreglo.
while (i < nL) {
    arreglo[ k ] = L[ i ];
    ++i;
    ++k;
}
while (j < nR) {
    arreglo[ k ] = R[ j ];
    ++j;
    ++k;
}
```

Heap Sort (int * arreglo, int tamano)

```
for (int i = tamano - 1; i >= 0; --i) {
    heapify(arreglo, tamano, i);
}
for (int j = tamano - 1; j >= 0; --j) {
    int aux = arreglo[0];
    arreglo[0] = arreglo[j];
    arreglo[j] = aux;
    heapify(arreglo, j, 0);
}
```

Heapify (int * arreglo, int tamano, int i)

```
int max;
int l = left(i);
int r = right(i);
if (l < tamano && arreglo[l] > arreglo[i]) {
    max = l;
} else {
    max = i;
}
if (r < tamano && arerglo[r] > arreglo[max]) {
    max = r;
}
if (max != i) {
    int aux = arreglo[i];
    arreglo[i] = arreglo[max];
    arreglo[max] = aux;
    heapify(arreglo, tamano, max);
}
```

Left (int i)

```
return (2*i)+1;
```

Right (int i)

```
return (2*i)+2;
```

Quick Sort (int * arreglo, int l, int r)

```
if (l < r) {
    int m = partition(arreglo, l, r);
    Quick Sort(arreglo, l, m - 1);
    Quick Sort(arreglo, m + 1, r);
}
```

Partition (int * arreglo, int l, int r)

```
int p = arreglo[r];
int i = l - 1;      int j;
for (j = l; j <= r - 1; ++j) {
    if (arreglo[j] <= p) {
        ++i;
        int aux1 = arregli[i];
        arreglo[i] = arreglo[j];
        arreglo[j] = aux1;
    }
}
int aux2 = arreglo[i+1];
arreglo[i+1] = arreglo[r];
arreglo[r] = aux2;
return i + 1;
```

Radix Sort (int * arreglo, int tamano)

```
fix (arreglo, tamano, true);
int max = getMax(arreglo, tamano);
for (int pos = 1; max/pos > 0; pos *= 10) {
    Count Sort(arreglo, tamano, pos)
}
Fix(arreglo, tamano, false);
```

Count Sort (int * arreglo, int l, int r)

```
int histo[tamano] = 0;
int histoAcu[k] = 0;
for (int k = 0; k < tamano; ++k) {
    histoAcu[k] = 0;
    histo[k] = 0;
}
for (int i = 0; i < tamano; ++i) {
    histoAcu[(arreglo[i]/pos) % 10]++;
}
for (int i = 1; i < tamano; ++i) {
    histoAcu[i] += histoAcu[i-1];
}
for (int i = tamano - 1; i >= 0; --i) {
    histo[ histoAcu[(arreglo[i]/pos % 10]-1] = arreglo[i];
    histoAcu[(arreglo[i]/pos) % 10]--
}
for (int i = 0; i < tamano; ++i) {
    arreglo[i] = histo[i];
}
```

Fix (int * arreglo, int tamano, bool x)

```
int min;
if (x) {
    min = arreglo[0];
    for (int i = 0; i < tamano, ++i) {
        if (arreglo[i] < min) {
            min = arreglo[i]; {
        }
    }
    if (min ¡ 0) {
        min *= -1;
    }
    for (int j = 0; j = < tamano; ++j) {
        arreglo[j] += min;          }
    this− >min = min;
} else {
    for (int k = 0; k < tamano; + + k) {
        arreglo[k] -= this− >min;
    }
}
```

getMax (int * arreglo, int tamano)

```
int max = arreglo[0];
for (int i = 0; i < tamano; ++i) {
    if (arreglo[i] = > max) {
        max = arreglo[i];
    }
}
return max;
```

Mediana (int * arreglo, int tamaño)

```
int mediana, x, aux;
int k = tamano/2;
int l = 0;
int r = tamano - 1;
int i = partition(arreglo, l, r);
while (k != i) {
    if (i < k) {
        i = partition(arreglo, i + 1, r);
    }
    if (i < k) {
        i = partition(arreglo, l, i - 1);
    }
}
mediana = arreglo[k];
return mediana;
```