Análisis de algoritmos de ordenamiento. Parte II

Fabián Orozco Chaves - B95690

Resumen—En este trabajo se realiza la implementación de seis algoritmos de ordenamiento utilizando C++ como lenguaje de programación, para comparar su eficiencia se mide el tiempo que tarda cada uno en ordenar un arreglo, de estos, cuatro de distinto tamaño. El resultado de su comparación es aproximado a lo que se puede inferir con lo estudiado en clase: con una notable diferencia entre ordenamientos en cuanto a eficiencia temporal; teniendo los seis un comportamiento próximo a su complejidad temporal.

I. Introducción

En este trabajo se buscan implementar seis algoritmos de ordenamiento: por selección, por inserción, por mezcla, por montículos, ordenamiento rápido y ordenamiento por residuos; además de comprobar sus diferencias de eficiencia (complejidad de tiempo) en la realidad.

El algoritmo de ordenamiento por selección consiste en encontrar el menor de todos los elementos del arreglo e intercambiarlo con el que está en la primera posición; después buscará el segundo menor de todo el arreglo y así sucesivamente. Complejidad temporal $T(n) = \Theta(n^2)$

Por otra parte, el algoritmo de ordenamiento por inserción ordena de incluso de una forma más natural: elige un elemento y verifica si el elemento es menor que su antecesor, continúa haciendo esto hasta que encuentre el caso contrario, donde simplemente coloca el elemento delante del valor menor que él. Complejidad temporal: $T(n) = \Theta(n^2)$

Como tercer algoritmo se tiene al ordenamiento por mezcla el cual utiliza una técnica denominada “divide y vencerás” la cual consiste en dividir el problema inicial (grande) en subproblemas (más pequeños) hasta llegar a un caso trivial, donde el ordenamiento se simplifica. De esta manera, el arreglo inicial se subdivide en dos arreglos que serán subdivididos de manera recursiva hasta ser un caso trivial; después se compararán los elementos de cada pareja de subarreglos para ordenarlos y por último unirlos por medio de la recursión (esta vez ordenados). Complejidad temporal: $T(n) = \Theta(n \log n)$

El cuarto algoritmo es el ordenamiento por montículos el cual utiliza una estructura de datos llamada montículos (para este, con montículos máximos). La propiedad fundamental de éstos es que su “padre” debe ser mayor a los dos hijos para cada subárbol del árbol principal, al proceso de poner los máximos como padres se llama monticularizar, una vez monticularizada la estructura, se ordenan los elementos extrayendo el elemento de la raíz e intercambiando con el último elemento, realizando la comparación para cumplir la propiedad fundamental en cada iteración. Complejidad temporal: $T(n) = \Theta(n \log n)$ ya que

está acotado superiormente e inferiormente con $(n \log n)$, suele ser mejor que ordenamiento por mezcla.

El quinto algoritmo es el de ordenamiento rápido, el cual utiliza una subrutina llamada partición la cual se llama recursivamente y su función es tomar un elemento pivote p (seleccionado al azar) y reubica los elementos de forma que los menores que p estén a su izquierda y los mayores a la derecha, aunque no estén necesariamente ordenados; devuelve la posición final en que se encuentra p. Este suele ser uno de los mejores algoritmos en su caso promedio: $T(n) = \Theta(n \log n)$ sin embargo tiene un peor caso de $T(n) = \Theta(n^2)$ que ocurre cuando el arreglo está ordenado. El tiempo se puede mejorar con alevosía: eligiendo un elemento aleatorio del arreglo, poniéndolo en la última posición y agarrándolo como pivote.

Ordenamiento rápido suele ser mejor que ordenamiento por montículos debido a un tema de hardware: se explota la memoria porque utiliza posiciones secuenciales, de esta forma utiliza menos páginas y en consecuencia utiliza mejor la caché, en cambio ordenamiento por montículos demanda un salto mayor.

Por último, el sexto algoritmo es el de ordenamiento por residuos, el cual tiene su nombre debido a que extrae el valor del bit menos significativo dividiendo sucesivamente el número entre la base y tomando su residuo; de esta forma se ordena primero la parte menos significativa y luego la más significativa. Utiliza un ordenamiento por conteo estable (elementos iguales quedan en el orden original) el cual ordena por medio de histogramas acumulativos (no por comparaciones). La complejidad temporal depende de cuál sea el máximo entre la cantidad de elementos (n) y la cantidad de contadores utilizados (k), teniendo así un tiempo lineal: $T(n) = \Theta(máx\{n, k\})$ siendo útil cuando k << n.

En teoría el ordenamiento por residuos hace un peor uso de la caché que el ordenamiento rápido sin embargo para conjuntos grandes no tienen mucha diferencia de eficiencia.

II. Metodología

Para lograr lo propuesto la comparación se realizará por medio del ordenamiento de cuatro arreglos de enteros de distinto tamaño: 50000, 100000, 150000 y 200000 inicializados con valores pseudoaleatorios positivos y negativos donde importará la cantidad de tiempo que le tomó a cada algoritmo ordenar cada arreglo. Para cada ordenamiento se realizaron pruebas que comprobaron que el ordenamiento fuera verdaderamente ascendente para todos los valores de cada arreglo.

Para realizar las comparaciones se realizaron varias ejecuciones (tres reportadas) donde los resultados de una no discrepaban en gran medida con la siguiente, es decir, hubo una constancia esperada en los resultados de las tres ejecuciones.

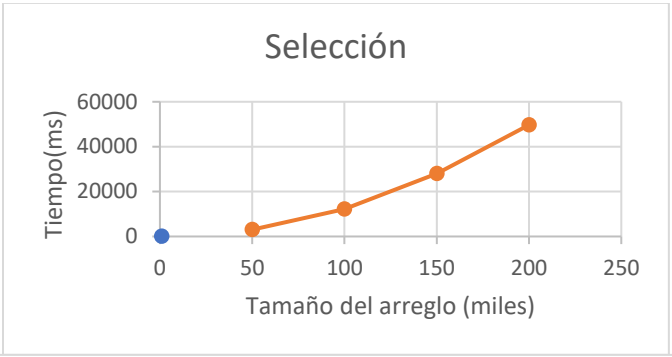
El código se muestra en el apéndice A y está basado en el pseudocódigo del libro de Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest y Clifford Stein titulado Introduction to Algorithms. 3ra edición.

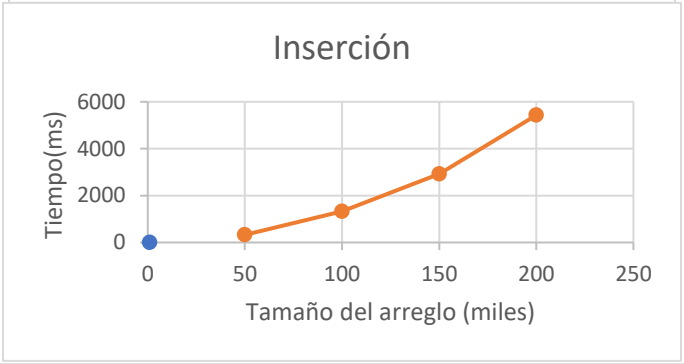
Cuadro I
TIEMPO DE EJECUCIÓN DE LOS ALGORITMOS.
TIEMPO(MS)

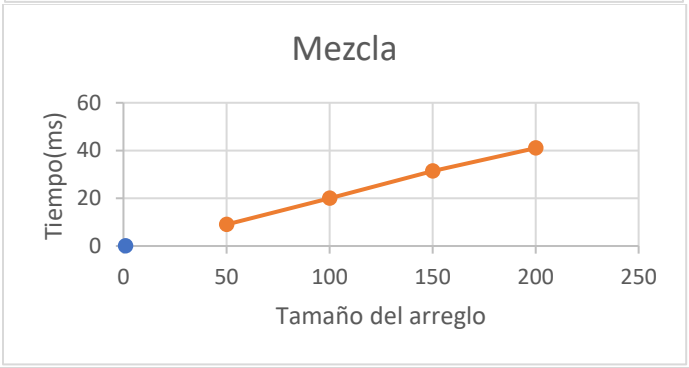
		EJECUCIONES			
ALGORITMO	Tamaño(k)	1	2	3	Promedio
SELECCIÓN	50	2958	3056	3094	3036
	100	12270	12065	12225	12186.7
	150	27590	28012	28602	28068
	200	48744	50226	50116	49695.3
INSERCIÓN	50	325	328	321	324.7
	100	1277	1358	1330	1321.7
	150	2860	2934	2970	2921.3
	200	5076	5448	5768	5430.7
MEZCLA	50	9	9	9	9
	100	20	20	20	20.0
	150	30	33	31	31.3
	200	40	41	42	41
MONTÍCULOS	50	2	3	2	2.5
	100	5	6	5	5.5
	150	10	10	10	10
	200	20	14	18	16
RÁPIDO	50	2	2	2	2
	100	5	6	5	5.3
	150	10	9	9	9.3
	200	12	12	13	12.3
RESIDUOS	50	0.49	0.57	1.1	0.7
	100	0.96	1	0.98	1
	150	1.12	1.12	1.12	1.1
	200	1.46	1.88	1.9	1.7

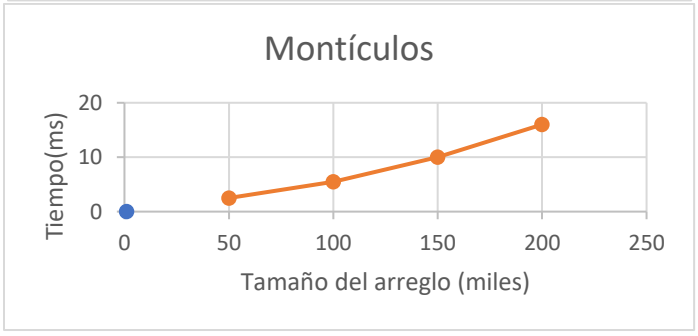
III. Resultados

Los tiempos de ejecución en milisegundos con extensión de dos decimales en las tres ejecuciones de los algoritmos se muestran en el cuadro I asociado a su respectivo tamaño de arreglo. A continuación, las gráficas de los tiempos promedio contra el tamaño del arreglo para cada uno de los algoritmos.









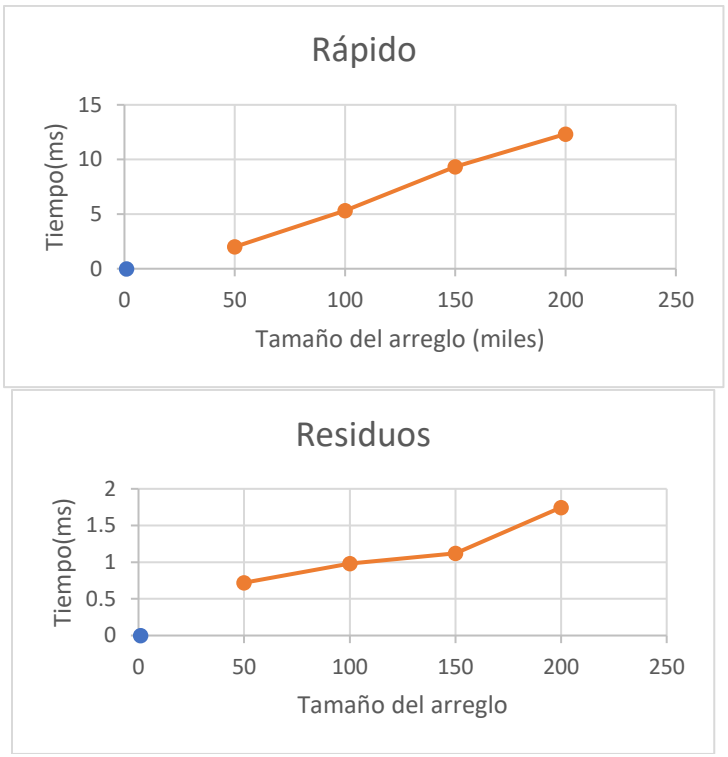


Figura 1. Tiempos promedio de ejecución de los algoritmos de ordenamiento por selección, inserción, mezcla, montículos, rápido y residuos.

La variación de los tiempos en las tres ejecuciones fueron similares, manteniendo un sentido acorde en cada una para los seis algoritmos, además se puede inferir el comportamiento esperado de cada algoritmo de ordenamiento, donde crecen según su complejidad temporal. $T(n) = \Theta(n^2)$ para el caso de ordenamiento por selección e inserción lo que indica que entre más grande sea un arreglo, su duración se verá altamente elevada (comportamiento aproximadamente parabólico) y $T(n) = \Theta(n \log n)$ en el caso de ordenamiento por mezcla, por montículos y ordenamiento rápido donde su duración en tiempo es casi lineal, dependiendo casi proporcionalmente al tamaño del arreglo (esto porque $n \log n < n^{1+\varepsilon}$ para $\varepsilon > 0$ y n suficientemente grande, además, el ordenamiento por residuos muestra un crecimiento muy reducido cuando crece el conjunto a ordenar, lo que demuestra que cumple lo esperado en teoría, teniendo una complejidad $T(n) = \Theta(máx\{n, k\})$.

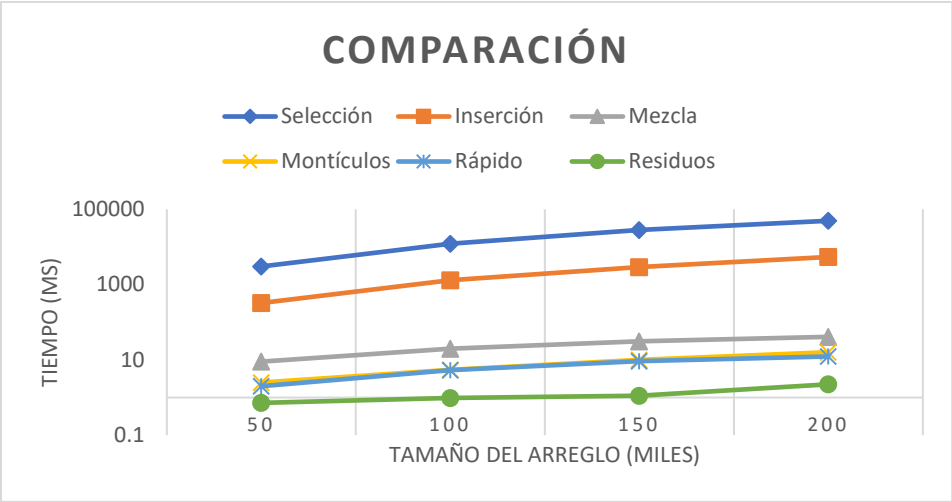


Figura 2. Gráfico comparativo de los tiempos promedio de ejecución de los algoritmos en escala logarítmica.

De la figura 2 se puede observar como el algoritmo de ordenamiento por residuos es significativamente mejor que los otros cinco algoritmos en cuanto a eficiencia temporal; además, se aprecia que la diferencia entre el algoritmo de selección e inserción es muy baja, manteniendo por poco una mejor eficiencia el algoritmo por selección que el de inserción. Otro detalle a observar es que tanto para ordenamiento por montículos como para ordenamiento rápido, los tiempos fueron sumamente parecidos, lo que indica que en la práctica se cumple lo que relata la teoría, dependiendo de un buen manejo de caché de uno u otro. La relación entre las seis curvas surgió como se esperaba en la teoría.

IV. Conclusiones

A partir de los resultados se concluye para arreglos de tamaño relativamente pequeños, tanto el algoritmo por inserción como el de selección tienen tiempos similares y aceptables pero a medida que crece el tamaño del arreglo se puede notar un decrecimiento en cuanto a eficiencia ya que el tiempo aumenta muchísimo, siendo el algoritmo por selección el menos eficiente, seguido del ordenamiento por inserción y mezcla, el ordenamiento por mezcla muestra un comportamiento distinto a los anteriores, el cual se mantiene casi constante independientemente del tamaño del arreglo; de la misma manera, el ordenamiento por montículos y el rápido comparten esta característica, manteniendo una duración casi constante con una mejor eficiencia. Por último, el ordenamiento que más destaca por su eficiencia es el ordenamiento por residuos, el cual tiene mejores tiempos que todos los anteriores, cumpliendo lo que se esperaba en teoría de una función lineal dependiente del tamaño del arreglo y/o cantidad de contadores.

Como se pudo inferir anteriormente y de acuerdo a lo estudiado en clase, el ordenamiento por residuos es el más eficiente de los seis tanto en teoría como en la práctica y queda plasmado en este trabajo que la eficiencia en teoría corresponde también a la realidad.

APÉNDICE A
CÓDIGO DE LOS ALGORITMOS

El código se muestra en los algoritmos 1, 2, 3, 4, 5 y 6 además el archivo adjunto a este documento contiene el código de los algoritmos y el pseudocódigo visto en clase basado en el libro de Cormen y colaboradores anteriormente mencionados.

Algoritmo 1 Algoritmo de ordenamiento por selección.

```
void seleccion (int *A, int n) {
for (int i = 0; i < n - 1; ++i) {
    int min = i;
    for (int j = i + 1; j < n; ++j) {
        if (A[j] < A[min]) min = j;
    }
    int aux = A[i];
    A[i] = A[min];
    A[min] = aux;
    }
}
```

Algoritmo 2 Algoritmo de ordenamiento por inserción

```
void insercion(int *A, int n) {
    int j, key, i;
    for (j=1; j < n; ++j) {
        key = A[j];
        i = j-1;
        while (i >= 0 && A[i] > key) {
            A[i+1] = A[i];
            --i;
       }
       A[i+1] = key;
    }
}
```

Algoritmo 3 Algoritmo de ordenamiento por mezcla (mergeSort)

```
void mergeSort(int* arr, int inicio, int fin){
  if (inicio < fin) {
    int mitad = (inicio + fin) / 2;
    mergeSort(arr, inicio, mitad);
    mergeSort(arr, mitad + 1, fin);
    merge(arr, inicio, mitad, fin);
  }
}
```

Nota: se utiliza `INT_MAX` como centinela el cual sustituye al valor ∞ utilizado en el libro, para ello se incluye la biblioteca `<limits.h>`.

Algoritmo 3.1 Algoritmo de ordenamiento por mezcla (merge)

```
void merge(int* arr, int izq, int mitad, int der) {
  const int subArr1 = mitad - izq + 1;
  const int subArr2 = der - mitad;
  int* izqArray = new int[subArr1 + 1];
  int* derArray = new int[subArr2 + 1];

  for (int i = 0; i < subArr1; ++i) {
      izqArray[i] = arr[izq + i];
  }

  for (int j = 0; j < subArr2; ++j) {
      derArray[j] = arr[mitad + 1 + j];
  }

  izqArray[subArr1] = INT_MAX;
  derArray[subArr2] = INT_MAX;
  int indexSubArr1 = 0;
  int indexSubArr2 = 0;

  for (int indexMergeArr = izq; indexMergeArr <= der; indexMergeArr++) {
      if (izqArray[indexSubArr1] <= derArray[indexSubArr2]) {
          arr[indexMergeArr] = izqArray[indexSubArr1];
          ++indexSubArr1;
      } else {
          arr[indexMergeArr] = derArray[indexSubArr2];
          ++indexSubArr2;
      }
  }
  delete [] izqArray;
  delete [] derArray;
}
```

Algoritmo 4 Algoritmo de ordenamiento por montículos (heapsort – max_heapify)

```
void max_heapify(int* A, int i, int& length){
    // L = LEFT(i)
    int L = (i << 0x1);
    // r = RIGHT(i)
    int r =(i << 0x1) | 0x1;
    int largest = 0;
    // if L <= A.heap-size and A[L] > A[i]
    if (L <= length && A[L] > A[i]) {
        // largest = L
        largest = L;
    }
    // else largest = i
    else {
        largest = i;
    }
    // if r <= A.heap-size and A[r] > A[largest]
    if (r <= length && A[r] > A[largest]) {
        // largest = r
        largest = r;
    }
    // if largest ¡= i
    if(largest != i) {
        // exchange A[i] with A[largest]
        int temp = A[largest];
        A[largest] = A[i];
        A[i] = temp;
        // MAX-HEAPIFY(A, largest)
        max_heapify(A, largest, length);
    }
}
```

Algoritmo 4.1 Algoritmo de ordenamiento por montículos (heapsort – build_max_heap)

```
void build_max_heap(int* A, int& length) {
    // A.heap-size = A.length
    int A_heap_size = length;
    // for i = rounddown(A.length/2) to 1
    for(int i = (A_heap_size/2); i > 0; --i) {
        // MAX-HEAPIFY(A, i)
        max_heapify(A, i, length);
    }
}
```

Algoritmo 4.2 Algoritmo de ordenamiento por montículos (heapSort)

```
void heapSort(int* A, int& length) {
    // BUILD-MAX-HEAP(A)
    build_max_heap(A, length);
    // for i = A.length downto 2
    for(int i = length; i > 1; --i) {
        // exchange A[1] with A[i]
        int temp = A[i];
        A[i] = A[1];
        A[1] = temp;
        // A.heap-size = A.heap-size - 1
        --length;
        // MAX-HEAPIFY(A, 1)
        max_heapify(A,1, length);
    }
}
```

Algoritmo 5 Algoritmo de ordenamiento rápido (quicksort - partition)

```
int partition(int* array, int inicio, int fin) {
    // x = A[r]
    int x = array[fin];
    //i = p - 1
    int i = inicio - 1;
    //for j = p to r - 1
    for (int j = inicio; j < fin; ++j) {
        // if A[j] <= x
        if (array[j] <= x) {
            // i = i + 1
            ++i;
            // exchange A[i] with A[j]
            int temp = array[j];
            array[j] = array[i];
            array[i] = temp;
        }
    }
    // exchange A[i+1] with A[r]
    int temp = array[fin];
    array[fin] = array[i+1];
    array[i+1] = temp;
    return i + 1;
}
```

Algoritmo 5.1 Algoritmo de ordenamiento rápido (quicksort)

```
void quickSort(int* array, int inicio, int fin) {
    // if p < r
    if (inicio < fin) {
        // q = PARTITION(A, p, r)
        int pivote = partition(array, inicio, fin);

        // QUICKSORT(A, p, q - 1)
        quickSort(array, inicio, pivote-1);

        // QUICKSORT(A, q + 1, r)
        quickSort(array, pivote+1, fin);
    }
}
```

Algoritmo 6 Algoritmos auxiliares de ordenamiento por residuos (radixsort)

```
int maximo(int* A, int size) {
    int max = A[0];
    for (int i = 0; i < size; ++i) { if (A[i] > max) max = A[i]; }
    return max;
}

void invert_array(int *A, int length) {
    int i, j, temp;
    for(i = 0, j = length-1; i < length/2; i++ , j--) {
        temp=A[i];
        A[i]=A[j];
        A[j]=temp;
    }
}

void cuenteSignos(int* A, int& length,int& qtyNegativos, int&
qtyPositivos) {
    for (int i = 0; i < length; ++i) { A[i] < 0 ? ++qtyNegativos :
++qtyPositivos; }
}
```

Algoritmo 6.1 Algoritmo de ordenamiento por residuos (radixsort – counting sort)

```
void countingSort(int* A, int* B, int k, int& length) {
    A--; B--;
    int* C = new int[k];
    for (int i=0; i < k; ++i) { C[i] = 0; }
    for(int j = 1; j <= length; ++j) { C[A[j]] = C[A[j]] + 1; }
    for(int i = 1; i < k; ++i) { C[i] = C[i] + C[i-1]; }
    for(int j = length; j > 0; --j) {
        B[C[A[j]]] = A[j];
        C[A[j]] = C[A[j]] - 1;
```

```
    }
}
```

Algoritmo 6.2 Algoritmo principal de ordenamiento por residuos (radixsort)

```
void radixsort(int* A, int n) {
    int qtyNegativos = 0, qtyPositivos = 0;
    cuenteSignos(A, n, qtyNegativos, qtyPositivos);

    int* negativos = new int[qtyNegativos];
    int* negOrdenados = new int[qtyNegativos];
    int* positivos = new int[qtyPositivos];
    int* posOrdenados = new int[qtyPositivos];

    int j = 0, k = 0;
    for(int i = 0; i < n; ++i) { // llena subarreglos (negativo y
positivo)
        A[i] < 0 ? negativos[j++] = -A[i] : positivos[k++] = A[i];
    }
    // ordena positivos
    int max = maximo(positivos, qtyPositivos);
    countingSort(positivos, posOrdenados, max+1, qtyPositivos);
    // ordena negativos
    max = maximo(negativos, qtyNegativos);
    countingSort(negativos, negOrdenados, max+1, qtyNegativos);
    invert_array(negOrdenados, qtyNegativos);
    // hace merge de negativos y positivos al principal
    int i = 0;
    for( ; i < qtyNegativos; ++i) { A[i] = -negOrdenados[i]; }
    for( k = 0 ; k < qtyPositivos; ++k){ A[i++] = posOrdenados[k]; }
}
```

REFERENCIAS

Cormen, T. H., Leiserson, C. E., Rivest, R. L.,, Stein, C. (2001). Introduction to Algorithms. The MIT Press. ISBN: 0262032937