**Algoritmos y Estructuras de Datos**

**Laboratorio 1 Unidad 1**

Nelson López.

Carlos Lizalda.

Santiago Chasqui.

**1. Identificación del problema:** Una empresa solicita el desarrollo de un programa que permita manipular un arreglo de números haciendo uso de distintas herramientas, las cuales deberán estar incluidas en el programa y que serán descritas con mayor detalle a continuación:

|  |  |
| --- | --- |
| R1 |  |
| Nombre | Permitir el ingreso de valores numéricos |
| Resumen | Se ha de permitir al usuario ingresar los valores del arreglo a ordenar, dichos valores pueden ser enteros o números de coma flotante. La extensión del arreglo puede ser tan grande como el usuario lo requiera. |
| Entradas | Los valores que constituyen el arreglo de números a ordenar. |
| Resultado | Se ha ingresado el arreglo de números a ordenar |

|  |  |
| --- | --- |
| R2 |  |
| Nombre | Generar valores numéricos de manera aleatoria |
| Resumen | Se debe permitir al usuario generar un arreglo de números elegidos al azar, pero teniendo en cuenta las especificaciones del usuario como la de mostrar o no valores repetidos y también verificando el rango en el que se deben generar los números. |
| Entradas | Especificaciones del usuario |
| Resultado | Se generó de manera aleatoria un arreglo de números y teniendo en cuenta las especificaciones ingresadas por el usuario. |

|  |  |
| --- | --- |
| R3 |  |
| Nombre | Generar números aleatorios ordenados ascendentemente |
| Resumen | Genera un arreglo de números ordenados de menor a mayor. El sistema debe ser responsable de hacer esta generación aleatoria entre los límites en que deba ser generado el conjunto de números. |
| Entradas | Límites de generación de números (máximo y mínimo) |
| Resultado | Se ha generado un arreglo de números aleatorios |

|  |  |
| --- | --- |
| R4 |  |
| Nombre | Generar números aleatorios ordenados descendentemente |
| Resumen | Genera un arreglo de números ordenados de mayor a menor. El sistema debe ser responsable de hacer esta generación aleatoria entre los límites en que deba ser generado el conjunto de números. |
| Entradas | Límites de generación de números (máximo y mínimo) |
| Resultado | Se ha generado un arreglo de números aleatorios descendentemente |

|  |  |
| --- | --- |
| R5 |  |
| Nombre | Generar número en orden completamente aleatorio |
| Resumen | Como se evidenció en los requerimientos anteriores (R3 y R4), el programa debía hasta ese punto generar un arreglo de números aleatorios, pero ordenados de cierta manera (ascendentemente o descendentemente), ahora el programa ha de ser capaz de generar los números aleatorios en un orden completamente aleatorio, es decir no deben estar ordenados de ninguna forma. |
| Entradas | Límites de generación de números (máximo y mínimo) |
| Resultado | Se ha generado una lista de números aleatorios sin ningún orden particular |

|  |  |
| --- | --- |
| R6 |  |
| Nombre | Desordenar cierta cantidad de un conjunto de números. |
| Resumen | Se desordena un porcentaje específico de la cantidad total de números ingresada. |
| Entradas | Porcentaje de números a desordenar. |
| Resultado | Se desordenó el porcentaje de números ingresado por el usuario en el conjunto de números. |

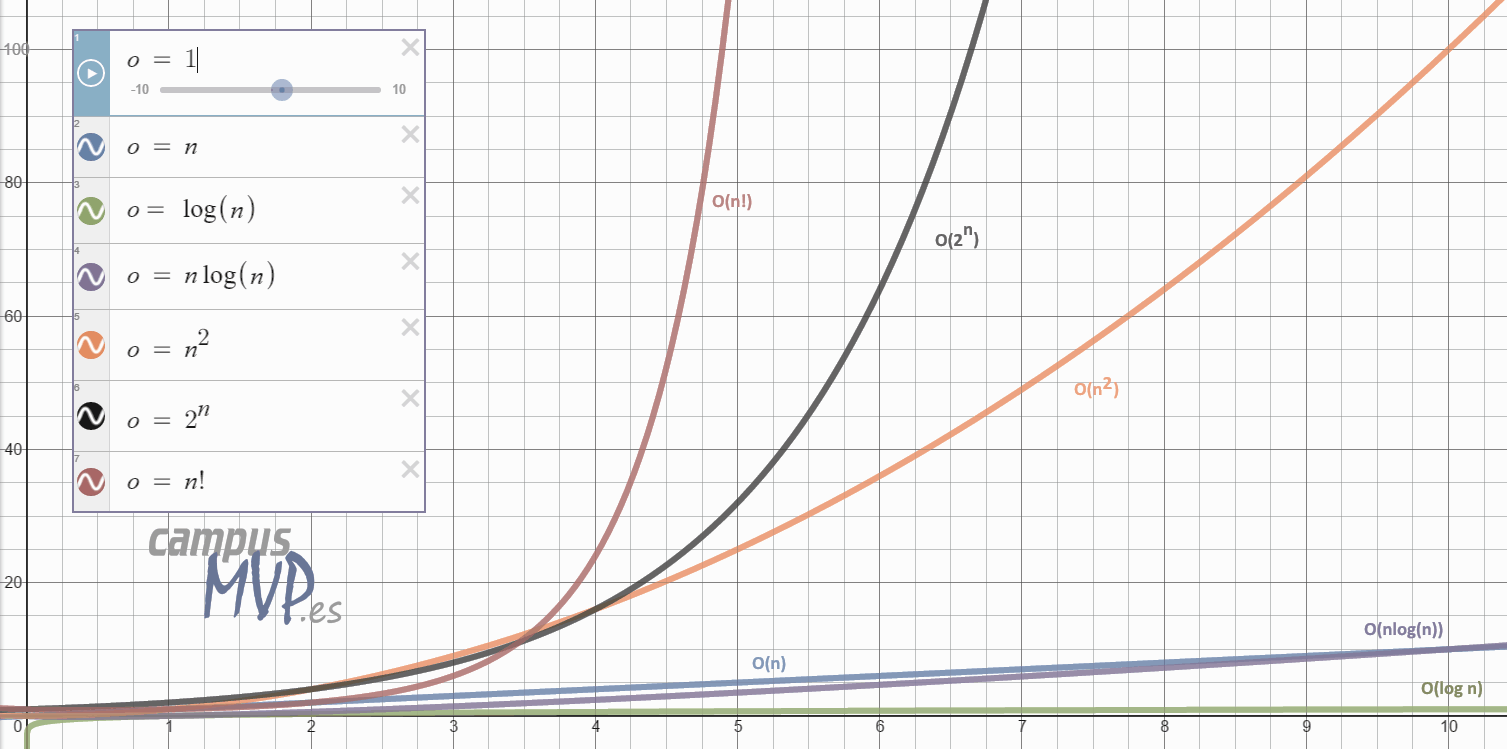
Requerimientos no funcionales:

* El programa debe permitir crear datos tanto de tipo int como de tipo float.
* Elegir el algoritmo de ordenamiento adecuado para el tipo de entrada ingresada.

**2. Recopilación de la información:** A continuación se describirán los conceptos formales necesarios para abordar el problema de manera pertinente:

* **Eficiencia:** Medida del uso de los recursos computacionales requeridos por la ejecución de un algoritmo en función del tamaño de las entradas.
* **Estabilidad:** Medida del uso de los recursos computacionales requeridos por la ejecución de un algoritmo en función del tamaño de las entradas.
* **Notación asintótica:** Estudian el comportamiento del algoritmo cuando el tamaño de las entradas es lo suficientemente grande, sin tener en cuenta lo que ocurre para entradas pequeñas y obviando factores constantes.

Para ilustrar la idea de la notación asintótica a la que se hace referencia en todo este documento, se presenta la gráfica de tiempo (eje y) vs número de entradas (eje x)

****

Fuente**:** https://www.campusmvp.es/recursos/post/Rendimiento-de-algoritmos-y-notacion-Big-O.aspx

* **Eficacia:** Hace referencia en la capacidad para alcanzar un objetivo, aunque en el proceso no se haya hecho el mejor uso de los recursos.
* **Estabilidad:** Se dice que un algoritmo de ordenamiento es estable cuando los elementos equivalentes que este pueda recibir como entrada, conservan su jerarquía de posición respecto de los demás elementos.
* **Algoritmo de ordenamiento:** Los algoritmos de ordenamiento nos permite, como su nombre lo dice, ordenar información de una manera especial basándonos en un criterio de ordenamiento definido previamente.A continuación se presentaran algunos de los algoritmos de ordenamiento más comunes:
  + **QuickSort:** Este es un algoritmo de ordenamiento de complejidad O(n logn) en el caso promedio y que es eficaz para todo tipo de números, pero con la desventaja de que en su peor caso la complejidad aumenta a O y se da cuando la lista de números se encuentra ordenada en su mayoría.
  + **Counting sort:** Es un algoritmo de ordenamiento de complejidad temporal O(n+k) con la ventaja de que conserva la misma complejidad (que de por sí ya es bastante eficiente) sin importar el caso presentado, pero con la desventaja de que solo ordena números enteros, ya que para otros tipos de números como los decimales, presenta deficiencias.
  + **Insertion sort:** Este algoritmo de ordenamiento de complejidad temporal O () es una de sus principales desventajas. Aunque permite ordenar cualquier tipo de números, utiliza un espacio muy mínimo en memoria y su método de ordenamiento resulta bastante intuitivo para comprender.
  + **Heapsort:** Heapsort es un algoritmo de ordenamiento con complejidad temporal O (n logn) que tiene la ventaja de utilizar una cantidad mínima en memoria, pero que resulta muy inestable para ciertos casos en particular, principalmente cuando el a lista de números a ordenar existen elementos repetidos.
  + **Merge Sort:** Es un algoritmo de ordenamiento con complejidad temporal O(n logn) y que permite ordenar cualquier lista de números reales, pero con la desventaja de que ocupa un espacio notable en memoria debido a su naturaleza recursiva.
  + **Shell Sort:** Este es un algoritmo de ordenamiento con complejidad temporal O ( ) que requiere de una cantidad mínima en memoria, pero que en la mayoría de los casos es poco eficiente.
  + **Selection Sort:** Este es un algoritmo de ordenamiento con complejidad temporal O) y consiste en encontrar el menor valor de un arreglo de números para luego intercambiarlo con el valor que se encuentra en la posición de un índice que recorre el arreglo. Este proceso se repite de manera reiterada hasta que se comparan todos los elementos.
  + **Bucket Sort**: Es un algoritmo de ordenamiento de complejidad temporal O(n) que consiste en dividir el arreglo de números que recibe como entrada en diferentes grupos de números con el fin de ordenar cada uno de ellos por separado para luego unirlos y finalmente mostrar el arreglo ordenado.
  + **Radix Sort**: Este es un algoritmo de ordenamiento de complejidad O(n) que consiste en acomodar un arreglo de números ordenando primero sus dígitos utilizando sistema de numeración decimal. Sólo sirve para enteros.
  + **Bitonic Sort:** Es un algoritmo de ordenamiento de complejidad O(n log)) que ordena un arreglo de datos de manera paralela, es decir, ordena de forma simultánea en diferentes lugares de la entrada.
  + **Comb Sort:** Este es un algoritmo de ordenamiento con complejidad temporal O(n logn) que se basa en el algoritmo Bubble sort, pero que lo optimiza al comparar elementos cuya distancia entre sí sea mayor a 1 posiciones.
  + **Pancake Sort:** Es un algoritmo de ordenamiento con complejidad temporal O(n^2) y consiste en ordenar un arreglo de números moviendo los elementos del arreglo hasta ordenarlos.

**3. Búsqueda de soluciones creativas:**

Existen numerosos algoritmos que se encargan de ordenar elementos bajo un criterio de ordenamiento específico y es por esto que resulta imprescindible analizar de manera detallada las características de cada uno de estos algoritmos, para así poder determinar el más óptimo teniendo en cuenta las condiciones iniciales del problema y finalmente comparando su eficiencia para el caso en cuestión. A continuación se presentaran algunas alternativas que, a consideración propia, son las más factibles:

**Alternativa 1: Quicksort.**

Este es un algoritmo de ordenamiento de complejidad O(n logn) en caso promedio, que es eficaz para todo tipo de números, pero con la desventaja de que en su peor caso la complejidad aumenta a O () y se da cuando la lista de números se encuentra ordenada en su mayoría.

**Alternativa 2: Counting sort.**

Es un algoritmo de ordenamiento de complejidad temporal O(n+k) con la ventaja de que conserva la misma complejidad (que de por sí ya es bastante eficiente) sin importar el caso presentado, pero con la desventaja de que solo ordena números enteros, ya que para otros tipos de números como los decimales, presenta deficiencias.

**Alternativa 3: Insertion sort.**

Este algoritmo de ordenamiento de complejidad temporal O (), la cual es una de sus principales desventajas, aunque permite ordenar cualquier tipo de números, utiliza un espacio muy mínimo en memoria y su método de ordenamiento resulta bastante intuitivo.

**Alternativa 4: Heapsort.**

Heapsort es un algoritmo de ordenamiento con complejidad temporal O (n logn) que tiene la ventaja de utilizar una cantidad mínima en memoria, pero que resulta muy inestable para ciertos casos en particular, principalmente cuando el a lista de números a ordenar existen elementos repetidos.

**Alternativa 5: Merge Sort.**

Es un algoritmo de ordenamiento con complejidad O(n logn) y que permite ordenar cualquier lista de números reales, pero con la desventaja de que ocupa un espacio notable en memoria debido a su naturaleza recursiva.

**Alternativa 6: Shell Sort.**

Este es un algoritmo de ordenamiento con complejidad O () que requiere de una cantidad mínima en memoria, pero que en la mayoría de los casos es poco eficiente.

**Alternativa 7: Radix Sort.**

Este es un algoritmo de ordenamiento de complejidad O(n) que consiste en acomodar un arreglo de números ordenando primero sus dígitos utilizando sistema de numeración decimal. Sólo sirve para numeros enteros.

**4. Transición de las ideas a los diseños preliminares**

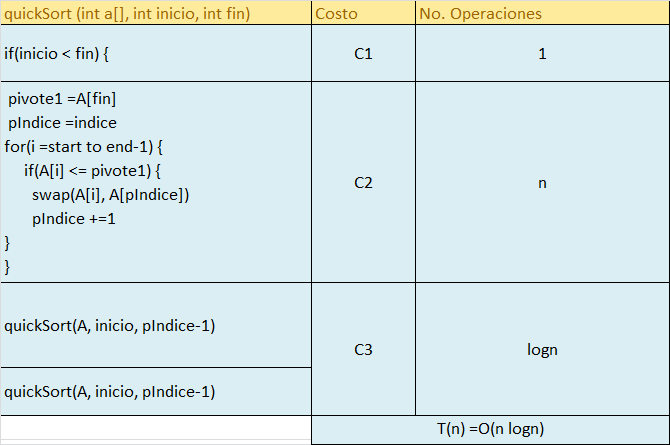
A continuación se evaluarán individualmente cada una de las alternativas planteadas en el inciso anterior y se procederá a descartar las propuestas menos factibles teniendo en cuenta los requerimientos del problema abordado.

Inicialmente se descartan los algoritmos de ordenamiento cuyas restricciones coinciden con casos que podrían darse dentro del problema abordado, es decir, casos en los que el algoritmo presentaría fallas a causa del tipo de entrada introducida. Este es el caso del algoritmo heapsort, el cual puede presentar fallas si en la lista de números a ordenar, se encuentran elementos repetidos.

El siguiente es el análisis de la complejidad espacial y temporal de cada uno de los algoritmos elegidos como alternativa:

**Alternativa 1: Quicksort.**

* Puede llegar a ser ineficiente cuando la lista se encuentra ordenada parcialmente.
* Es un algoritmo que en el caso promedio llega a ser muy eficiente. Su complejidad es O(n logn ) lo cual podría ser muy útil cuando las entradas están con un desorden relativamente alto (se habilitará una opción en la cual el usuario podrá elegir el porcentaje al cual quiere que sus datos estén desordenados).
* Se prefiere frente al mergesort para arreglos, debido a que ocupa menos espacio y tiempo. Esto se debe a que el merge ocupa un vasto espacio y a la hora de realizar las asignaciones de espacio se puede demorar un poco más.
* Permite ordenar cualquier tipo de números, lo que lo hace una opción potencialmente aceptable, teniendo en cuenta que el programa aceptará tanto naturales como de coma flotante.



Caracterización de la entrada: n es la cantidad de posiciones de A, n= A.length.

Tabla 1.1: Análisis de complejidad temporal del algoritmo Quicksort.

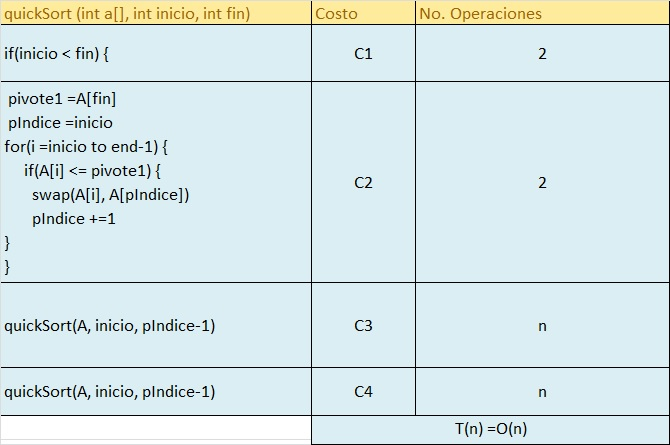
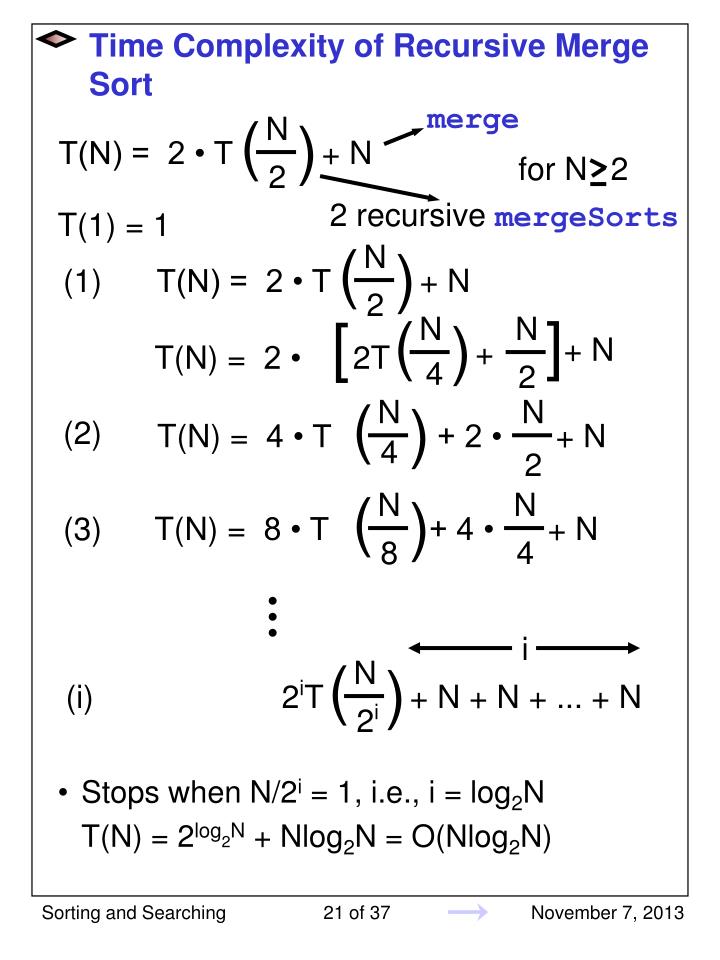


Tabla 1.2: Análisis de complejidad espacial del algoritmo quicksort.

**Alternativa 2: Merge Sort**

* Es un algoritmo con una muy buena complejidad temporal (O(n logn) ), lo que permite cubrir la necesidad de hacer las operaciones rápidamente.
* El ordenamiento puede realizarse con cualquier tipo de entrada (Naturales o de coma flotante).
* Al ser de naturaleza recursiva, ocupa un amplio espacio en memoria. La ventaja que se tiene frente a este inconveniente es que el cliente ha superpuesto la eficiencia temporal antes que la espacial. Además un coprocesador no comparte memoria con procesos del computador que puedan afectar el rendimiento.
* Es un algoritmo muy estable, es decir, no importa qué tan ordenada esté la entrada, va a tomar un tiempo similar.

****

*Análisis realizado por Yaphet****.*** [***https://www.slideserve.com/yaphet/sorting-and-searching***](https://www.slideserve.com/yaphet/sorting-and-searching)

Tabla 1.3: Análisis de complejidad temporal del algoritmo Merge sort

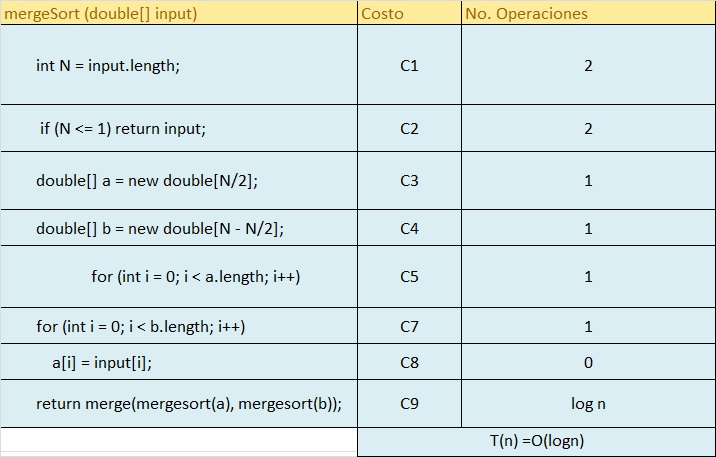
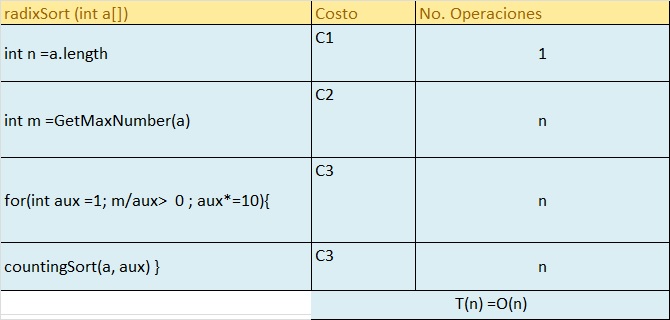
**

Tabla 1.4: Análisis de complejidad espacial del algoritmo Merge sort

**Alternativa 3: Radix Sort**

* Este algoritmo presenta una de las mejores complejidades que se puedan encontrar (O(n)), así que la rapidez está asegurada con este algoritmo.
* Utiliza el Counting Sort como subrutina de ordenamiento, lo cual hace que la complejidad espacial y temporal sea mayor. Pese a implementar otro algoritmo, su complejidad temporal sigue siendo lineal, debido a que la complejidad del counting es también lineal.
* Se puede ordenar cualquier tipo de números.



Caracterización de la entrada: n es la cantidad de posiciones de A, n= A.length.

Tabla 1.5: Análisis de complejidad temporal del algoritmo Radix sort.

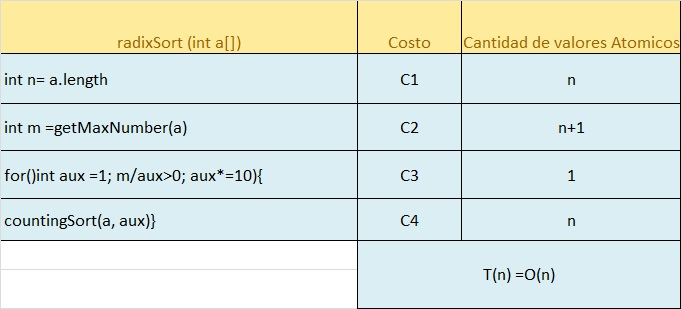
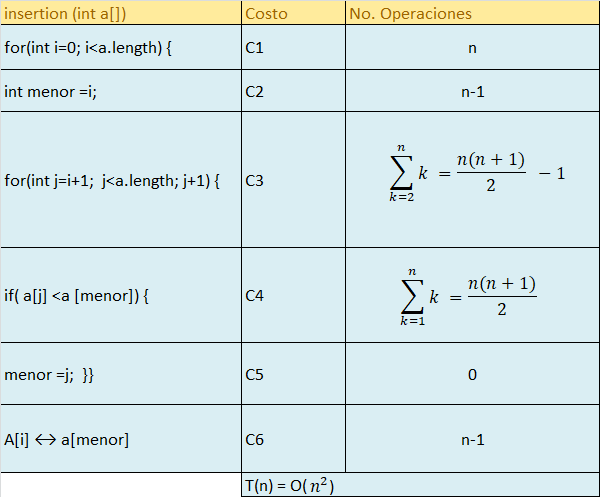


Tabla 1.6: Análisis de complejidad espacial del algoritmo Radix sort.

**Alternativa 4: Insertion sort**

* Es muy efectivo para entradas pequeñas a medianas.
* Sólo acepta números enteros.
* Es un algoritmo muy fácil de comprender.



Caracterización de la entrada: n es la cantidad de posiciones de A, n= A.length.

Tabla 1.7: Análisis de complejidad temporal del algoritmo Insertion sort.

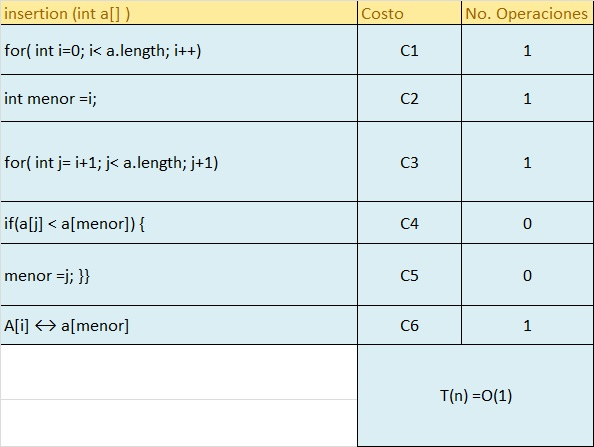


Tabla 1.8: Análisis de complejidad espacial del algoritmo Insertion sort.

**Alternativa 5: Counting Sort**

* Es un algoritmo bastante estable.
* Tiene la desventaja de que se puede utilizar únicamente para ordenar enteros.

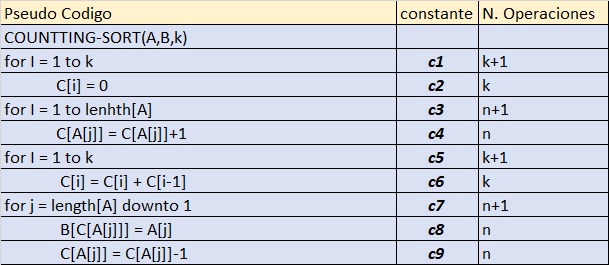
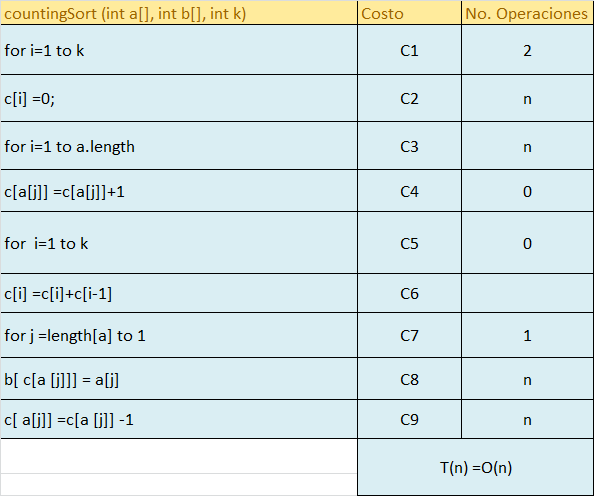


Tabla 1.9: Análisis de complejidad temporal del algoritmo Counting sort.



Caracterización de la entrada: n es la cantidad de posiciones de A, n= A.length.

Tabla 1.10: Análisis de complejidad temporal del algoritmo Counting sort.

Teniendo en cuenta el análisis anterior, resulta necesario descartar los algoritmos cuya complejidad temporal es elevada, como es el caso de los siguientes: Selection sort, Pancake sort y Shell sort. Estos son algoritmos que presentan una complejidad O (), lo cual hace que para entradas arbitrariamente grandes, sean supremamente ineficientes y por lo tanto no satisfacen de manera óptima las funcionalidades del problema.

Por último, es necesario evaluar cada uno de los aspectos de las alternativas faltantes para tener una idea más clara de cuáles serán las alternativas apropiadas para elegir:

**5. Evaluación y Selección de la Mejor Solución**

A continuación se expondrán los criterios mediante los cuales se elegirán los algoritmos de ordenamiento que permitan solucionar de manera óptima el problema planteado.

**Criterio 1:** Complejidad temporal del algoritmo de ordenamiento.

|  |  |
| --- | --- |
| **Complejidad Temporal** | **Puntuación** |
| O (n logn) | 8 |
| O (n+k) | 10 |
| O (n^2) | 3 |

**Criterio 2: Complejidad** espacial del algoritmo de ordenamiento.

|  |  |
| --- | --- |
| **Complejidad Espacial** | **Puntuación** |
| O (1) | 6 |
| O (n) | 5 |
| O (n logn) | 2 |

**Criterio 3:** Generalidad o especificidad del algoritmo de ordenamiento.

|  |  |
| --- | --- |
| **Tipo de algoritmo de ordenamiento** | **Puntuación** |
| General | 2 |
| Específico | 1 |

**Criterio 4:** Estabilidad

|  |  |
| --- | --- |
| **Estabilidad del algoritmo** | **Puntuación** |
| Estable | 2 |
| Inestable | 0 |

En el siguiente recuadro se mostraran las diferentes alternativas que, después de evaluarlas con los criterios descritos, se elegirán las que cumplan con mayor rigurosidad las condiciones impuestas.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **Criterio 1** | **Criterio 2** | **Criterio 3** | **Criterio 4** | **Total** |
| Alternativa 1. Quicksort | 8 | 5 | 2 | 2 | 17 |
| Alternativa 2. Merge Sort | 8 | 5 | 2 | 2 | 17 |
| Alternativa 3. Radix Sort | 10 | 5 | 1 | 2 | 18 |
| Alternativa 4. Insertion Sort | 3 | 6 | 2 | 2 | 13 |
| Alternativa 5. Counting Sort | 10 | 5 | 1 | 0 | 16 |

**Selección definitiva:** Teniendo en cuenta el método de selección utilizado anteriormente, se llegó a la conclusión de que los algoritmos que deberán ser usados en el software a crear son los algoritmos de ordenamiento Quick sort, Merge sort y Radix sort, debido a que estos son los algoritmos que solucionan el problema propuesto y que, a criterio propio, lo hacen de la manera mas optima y eficiente comparada con otros algoritmos.

**6. Preparación de informes y especificaciones:**

**Especificación del problema:**

**· Problema:** Ordenar un arreglo de números bajo diferentes criterios.

**· Entrada:** Arreglo de números a ordenar.

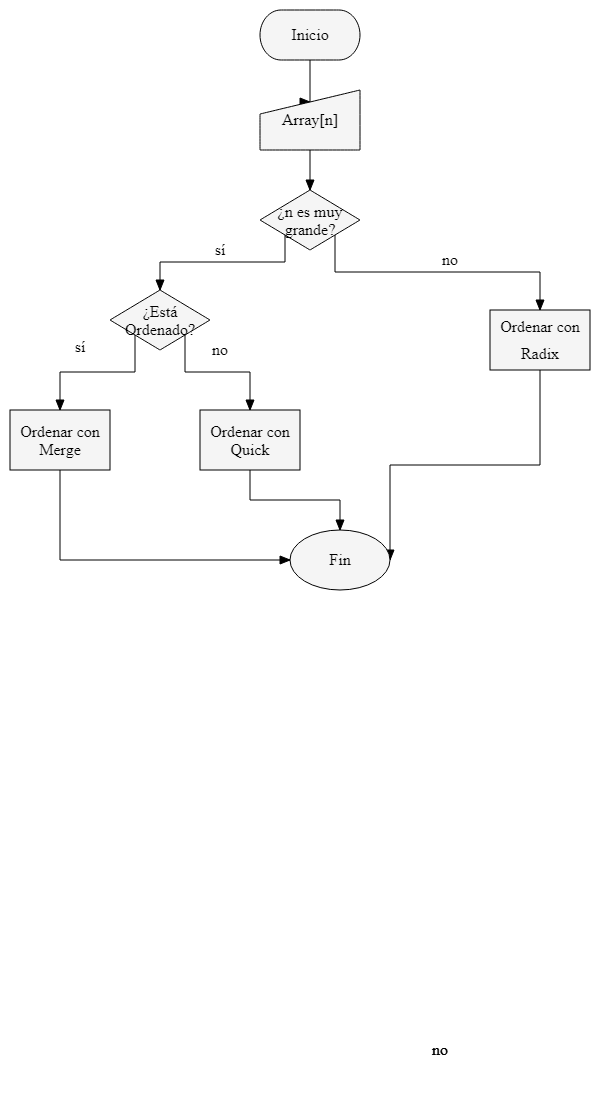
**· Salida:** Arreglo de números organizado de acuerdo a la condición de ordenamiento.

**Consideraciones:**

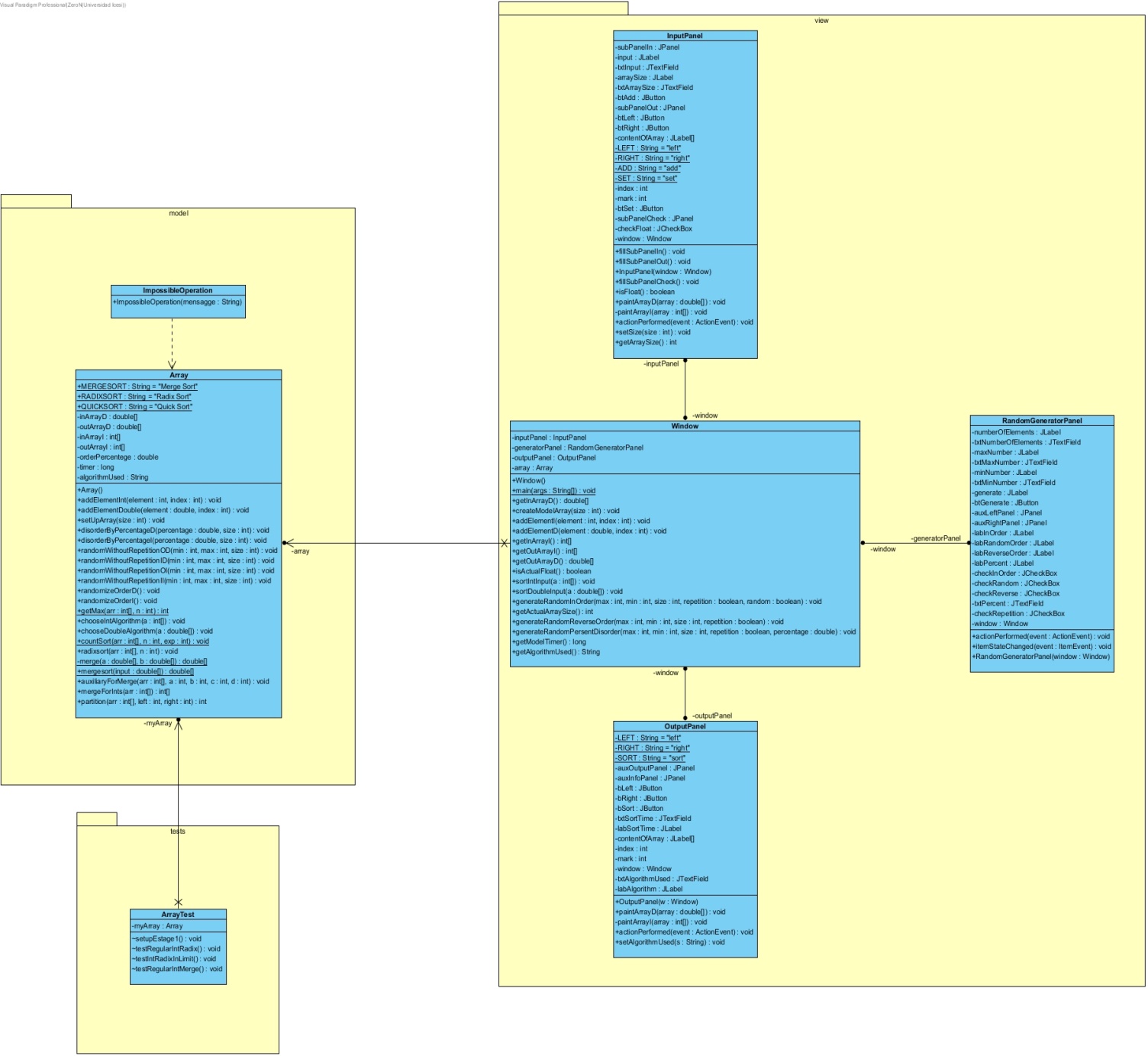
Se deben tener en cuenta las siguientes situaciones:

* El programa podrá elegir entre ordenar una instancia de entrada mediante un algoritmo u otro, para mejorar la eficiencia temporal y evitar posibles fallos a la hora de ordenar debidos a la inestabilidad de ciertos algoritmos.
* Para elegir el algoritmo a utilizar se tendrán en cuenta las características del arreglo ingresado como entrada, tales como: El intervalo en el que se genera, el porcentaje de desorden, el número de elementos.

A continuación se mostrará un diagrama de flujo que refleja de forma general el funcionamiento del programa y la manera en que éste operará de acuerdo a las diferentes condiciones a las que se verá enfrentado:



Adicional a ello se elaboró una representación de la arquitectura del software y las partes que lo componen, todo ello con el fin de tener una visión más clara de lo que va a ser el programa.



**7. Implementación del Diseño:**

A continuación se presentara una lista de las tareas a implementar por el programa:

1. Ordenar una lista de números.
2. Crear un arreglo de números de manera aleatoria.
3. Desordenar parte de una lista de números.

|  |  |
| --- | --- |
| **Especificación de subrutinas** | **Construcción** |
| |  |  | | --- | --- | | **Nombre** | randomWithoutRepetition | | **Descripción** | Este es un método que genera un arreglo de tipo double garantizando que ninguno de los elementos se va a repetir y organizándolos de manera ascendente. | | **Entrada** | Tamaño del arreglo a crear, rango del arreglo, es decir, el menor y mayor valor que debe aparecer en este. | | **Retorno** | Un arreglo de tipo double con las condiciones iniciales especificadas por el usuario. | | |  | | --- | | public void randomWithoutRepetitionOD(int min, int max, int size) {    inArrayD = new double[size];    double interval = (max-min) / size;    for (int i = 0; i < inArrayD.length; i++) {  inArrayD[i] =( Math.random() \* interval ) + (min + (interval \* i)) ;  }  } | |  |  | |  |  | |  |  | |  |  | |  |  | |  |  | |  |  | |  |  | |  |  | |  |  | |
| |  |  | | --- | --- | | **Nombre** | randomWithoutRepetitionID | | **Descripción** | Este es un método que genera un arreglo de tipo double garantizando que ninguno de los elementos se va a repetir y organizándolos de manera descendente | | **Entrada** | Tamaño del arreglo a crear, rango del arreglo, es decir, el menor y mayor valor que debe aparecer en este. | | **Retorno** | Un arreglo de tipo double con las condiciones iniciales especificadas por el usuario. | | public void randomWithoutRepetitionID(int min, int max, int size) {    inArrayD = new double[size];  double interval = (max-min) / size;    for (int i = inArrayD.length - 1; i >= 0 ; i++) {  inArrayD[i - inArrayD.length - 1] =( Math.random() \* interval ) + (min + (interval \* i)) ;  }  } |
| |  |  | | --- | --- | | **Nombre** | randomWithoutRepetitionOI | | **Descripción** | Este es un método que genera un arreglo de tipo int garantizando que ninguno de los elementos se va a repetir y organizándolos de manera ascendente. | | **Entrada** | Tamaño del arreglo a crear, rango del arreglo, es decir, el menor y mayor valor que debe aparecer en este. | | **Retorno** | Un arreglo de tipo int con las condiciones iniciales especificadas por el usuario. | | public void randomWithoutRepetitionOI(int min, int max, int size) {    inArrayI = new int[size];  double interval = (max-min) / size;    for (int i = 0; i < inArrayI.length; i++) {  inArrayI[i] = (int) (( Math.random() \* interval ) + (min + (interval \* i))) ;  }    } |
| |  |  | | --- | --- | | **Nombre** | randomWithoutRepetitionII | | **Descripción** | Este es un método que genera un arreglo de tipo int garantizando que ninguno de los elementos se va a repetir y organizándolos de manera descendente | | **Entrada** | Tamaño del arreglo a crear, rango del arreglo, es decir, el menor y mayor valor que debe aparecer en este. | | **Retorno** | Un arreglo de tipo int con las condiciones iniciales especificadas por el usuario. | | public void randomWithoutRepetitionII(int min, int max, int size) {    inArrayI = new int[size];  double interval = (max-min) / size;    for (int i = inArrayI.length - 1; i >= 0 ; i++) {  inArrayI[i - inArrayI.length - 1] =(int) (( Math.random() \* interval ) + (min + (interval \* i))) ;  }    } |
| |  |  | | --- | --- | | **Nombre** | randomizeOrderD | | **Descripción** | Este método se encarga de reorganizar de manera aleatoria los elementos de un array de tipo double. | | **Entrada** | Arreglo de tipo double. | | **Retorno** | Arreglo de tipo double reorganizado aleatoriamente. | | public void randomizeOrderD() {  for (int i = 0; i < inArrayD.length; i++) {  int num = (int) (Math.random() \* inArrayD.length);  double temp = inArrayD[i];  inArrayD[i] = inArrayD[num];  inArrayD[num] = temp;  }  } |
| |  |  | | --- | --- | | **Nombre** | randomizeOrderI | | **Descripción** | Este método se encarga de reorganizar de manera aleatoria los elementos de un array de tipo int. | | **Entrada** | Arreglo de tipo int. | | **Retorno** | Arreglo de tipo int reorganizado aleatoriamente. | | public void randomizeOrderI() {  for (int i = 0; i < inArrayI.length; i++) {  int num = (int) (Math.random() \* inArrayD.length);  int temp = inArrayI[i];  inArrayI[i] = inArrayI[num];  inArrayI[num] = temp;  }  } |
| |  |  | | --- | --- | | **Nombre** | getMax | | **Descripción** | Este método se encarga de encontrar el máximo valor en un arreglo de tipo int | | **Entrada** | Arreglo de tipo int | | **Retorno** | Numero mas grande encontrado dentro del arreglo. | | static int getMax(int arr[], int n)  {  int mx = arr[0];  for (int i = 1; i < n; i++)  if (arr[i] > mx)  mx = arr[i];  return mx;  } |
| |  |  | | --- | --- | | **Nombre** | chooseIntAlgorithm | | **Descripción** | Este método se encarga de elegir el algoritmo de ordenamiento adecuado para determinado arreglo de tipo int ingresado como parámetro. | | **Entrada** | Arreglo de tipo int | | **Retorno** | Organiza con el algoritmo adecuado el arreglo ingresado. | | public void chooseIntAlgorithm(int[] a) {    if(a.length <= 10000) {  int myIntArray[] = new int[a.length];  for (int i = 0; i < a.length; i++) {  myIntArray[i] = (int) a[i];  }    outArrayI = radixsort(myIntArray, myIntArray.length);  outArrayI = myIntArray.clone();  radixsort(outArrayI, outArrayI.length);  } } |
| |  |  | | --- | --- | | **Nombre** | countSort | | **Descripción** | Este es un algoritmo de ordenamiento que se encarga de organizar de manera ascendente un arreglo de tipo int. | | **Entrada** | Arreglo de tipo int a organizar, entero que indica el máximo número del arreglo | | **Retorno** | Arreglo ordenado. | | static void countSort(int arr[], int n, int exp)  {  int output[] = new int[n]; // output array  int i;  int count[] = new int[10];  Arrays.fill(count,0);    for (i = 0; i < n; i++)  count[ (arr[i]/exp)%10 ]++;    for (i = 1; i < 10; i++)  count[i] += count[i - 1];    for (i = n - 1; i >= 0; i--)  {  output[count[ (arr[i]/exp)%10 ] - 1] = arr[i];  count[ (arr[i]/exp)%10 ]--;  }    for (i = 0; i < n; i++)  arr[i] = output[i];    } |
| |  |  | | --- | --- | | **Nombre** | radixsort | | **Descripción** | Este es un algoritmo de ordenamiento que se encarga de organizar un arreglo de tipo int de manera ascendente | | **Entrada** | Arreglo de tipo int, tamanho del arreglo. | | **Retorno** | Arreglo ordenado. | | public void radixsort(int arr[], int n)  {  int m = getMax(arr, n);  for (int exp = 1; m/exp > 0; exp \*= 10)  countSort(arr, n, exp);  } |
| |  |  | | --- | --- | | **Nombre** | merge | | **Descripción** | Este es un algoritmo de ordenamiento que se encarga de organizar un arreglo de tipo double de tamaño relativamente pequeño. | | **Entrada** | Dos mitades de un arreglo de tipo double. | | **Retorno** | Arreglo ordenado. | | private static double[] merge(double[] a, double[] b) {  double[] c = new double[a.length + b.length];  int i = 0, j = 0;  for (int k = 0; k < c.length; k++) {  if (i >= a.length) c[k] = b[j++];  else if (j >= b.length) c[k] = a[i++];  else if (a[i] <= b[j]) c[k] = a[i++];  else c[k] = b[j++];  }  return c;  } |
| |  |  | | --- | --- | | **Nombre** | quick | | **Descripción** | Este es un algoritmo de ordenamiento que se encarga de ordenar un arreglo de tipo int ingresado como parámetro. | | **Entrada** | Arreglo de tipo int, primer elemento del arreglo, último elemento del arreglo. | | **Retorno** | Organiza con el algoritmo adecuado el arreglo ingresado. | | public void quick(int[] arr, int first, int last) {  int i = first;  int j = last;    int pivot = arr[first+(last-first)/2];    while(i<=j) {  while (arr[i] < pivot) {  i++;  }  while (arr[j] > pivot) {  j--;  }  if (i <= j) {  interchangeNumbers(arr, i, j);  i++;  j--;  }  }    if (first < j)  quick(arr, first, j);  if (i < last)  quick(arr, i, last);  } |

**Fuentes:**

[**https://www.geeksforgeeks.org/bucket-sort-2/**](https://www.geeksforgeeks.org/bucket-sort-2/)

[**https://gist.github.com/yeison/5606963**](https://gist.github.com/yeison/5606963)

<http://lwh.free.fr/pages/algo/tri/tri_es.htm>

<http://elvex.ugr.es/decsai/algorithms/slides/2%20Eficiencia.pdf>

[**http://informatica.wikia.com/wiki/Eficacia\_y\_eficiencia**](http://informatica.wikia.com/wiki/Eficacia_y_eficiencia)

<https://pereiratechtalks.com/analisis-de-algoritmos-de-ordenamiento/>

<https://www.slideserve.com/yaphet/sorting-and-searching>