Contexto problemático

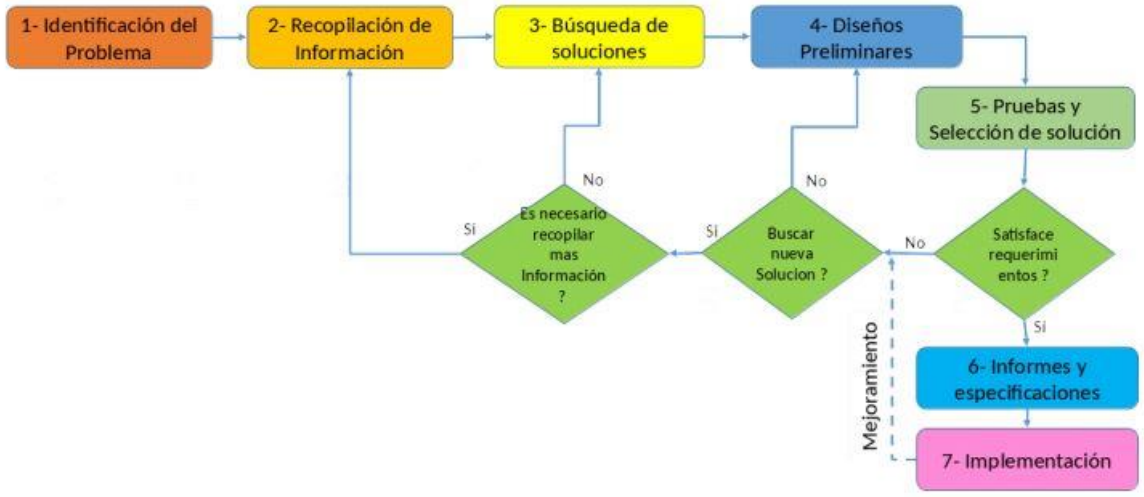
Se presenta un programa el cual presta el servicio de ordenamiento a arreglos de números con diferentes tamaños y tipos. La empresa después de estudiar los costos de implementación del ordenamiento como una operación nativa del coprocesador, ha decido implementar tres algoritmos diferentes de ordenamiento que permitan ordenar, muy rápidamente, números enteros de tamaño arbitrariamente grande y números en formato de coma flotante de cualquier tamaño. Se requiere soluciones para encontrar el algoritmo de funcionamiento más eficiente.

Desarrollo de la Solución

Para resolver la situación anterior se eligió el Método de la Ingeniería para desarrollar la solución siguiendo un enfoque sistemático y acorde con la situación problemática planteada.

Con base en la descripción del Método de la Ingeniería del libro “Introduction to Engineering” de Paul Wright,

Se definió el siguiente diagrama de flujo, cuyos pasos seguiremos en el desarrollo de la solución.



Paso 1. Identificación del Problema

Se reconocen de manera concreta las necesidades propias de la situación problemática así como sus síntomas y condiciones bajo las cuales debe ser resuelta.

Identificación de necesidades y síntomas:

- Existen una gran variedad de algoritmos que se comportan de manera diferente y no se sabe cuál se puede acoplar al problema.
- La solución debe garantizar que el algoritmo de ordenamiento es el más eficiente
- Se debe identificar un algoritmo que sea capaz de ordenar eficientemente los diferentes modos de desorden alternativos.

Definición del Problema

La empresa requiere del desarrollo de un algoritmo de ordenamiento que sea capaz de organizar eficaz y eficientemente una cantidad definida de números.

Paso 2. Recopilación de Información

Con el fin de tener una total claridad de los conceptos que aparecen en el problema se procede hacer una búsqueda de las definiciones de los términos estrechamente relacionados con el problema planteado.

Definiciones:

Fuente:

https://es.wikipedia.org/

Algoritmos de ordenamiento:

En computación y matemáticas un algoritmo de ordenamiento es un algoritmo que pone elementos de una lista o un vector en una secuencia dada por una relación de orden, es decir, el resultado de salida ha de ser una permutación —o reordenamiento— de la entrada que satisfaga la relación de orden dada.

Eficacia y Eficiencia:

Eficacia es la capacidad de lograr un efecto deseado, esperado o anhelado. En cambio, **Eficiencia** es la capacidad de lograr ese efecto en cuestión con el mínimo de recursos posibles o en el menor tiempo posible.

Tipos de datos numéricos:

En ciencias de la computación, un tipo de dato informático es un atributo de los datos que indica al ordenador (y/o al programador/programadora) sobre la clase de datos que se va a manejar. Esto incluye imponer restricciones en los datos, como qué valores pueden tomar y qué operaciones se pueden realizar.

El tipo de dato numérico puede ser real o entero, dependiendo del tipo de dato que se vaya a utilizar.

Enteros: son los valores que no tienen punto decimal, pueden ser positivos o negativos y el cero.

Tipo Numérico	Descripción	Tamaño	Envoltorio
byte	Entero con signo	8 bits	Byte
short	Entero con signo	16 bits	Short
int	Entero con signo	32 bits	Integer
long	Entero con signo	64 bits	Long
float	Real simple precisión	32 bits	Float
double	Real simple precisión	64 bits	Double

Estabilidad de un algoritmo:

Los algoritmos de ordenamiento estable mantienen un relativo preorden total. Esto significa que un algoritmo es estable solo cuando hay dos registros R y S con la misma clave y con R apareciendo antes que S en la lista original.

Los algoritmos de ordenamiento inestable pueden cambiar el orden relativo de registros con claves iguales, pero los algoritmos estables nunca lo hacen. Los algoritmos inestables pueden ser implementados especialmente para ser estables. Una forma de hacerlo es extender artificialmente el cotejamiento de claves, para que las comparaciones entre dos objetos con claves iguales sean decididas usando el orden de las entradas original.

Algoritmo "In-place":

En ciencias de la computación, un algoritmo in-place es un algoritmo el cual transforma una entrada sin usar una estructura auxiliar de datos. Sin embargo una pequeña cantidad de memoria extra está disponible para las estructuras de datos. La entrada esta usualmente sobrescrita por una salida como el algoritmo que lo ejecuta. El algoritmo in-place actualiza secuencias de entradas solo mediante el remplazo o el cambia de elementos. Un algoritmo que no es in-place se llama "not-in-place" o "out-of-place".

Paso 3. Búsqueda de Soluciones Creativas

Para este paso, buscamos en varias fuentes especializadas diversas estrategias a través de las cuales puedan ser encontradas diferentes algoritmos de ordenamiento que den solución al problema planteado. Los métodos encontrados son los siguientes:

Alternativa 1. Algoritmo de selección:

Los pasos del algoritmo son:

1. Seleccionar el elemento más pequeño de la lista A; intercambiarlo con el primer elemento A[0]. Ahora la entrada más pequeña está en la primera posición del vector.

2. Considerar las posiciones de la lista A[1], A[2], A[3]..., seleccionar el elemento más pequeño e intercambiarlo con A[1]. Ahora las dos primeras entradas de A están en orden.

3. Continuar este proceso encontrando o seleccionando el elemento más pequeño de los restantes elementos de la lista, intercambiándolos adecuadamente.

Alternativa 2. Algoritmo de inserción:

El algoritmo correspondiente a la ordenación por inserción contempla los siguientes pasos:

1. El primer elemento A[0] se considera ordenado; es decir, la lista inicial consta de un elemento.

2. Se inserta A[1] en la posición correcta, delante o detrás de A[0], dependiendo de que sea menor o mayor.

3. Por cada bucle o iteración i (desde i=1 hasta n-1) se explora la sublista A[i-1] . . A[0] buscando la posición correcta de inserción; a la vez se mueve hacia abajo (a la derecha en la sublista) una posición todos los elementos mayores que el elemento a insertar A[i], para dejar vacía esa posición.

4. Insertar el elemento a la posición correcta.

Alternativa 3. Algoritmo de Burbuja:

En el caso de un array (lista) con n elementos, la ordenación por burbuja requiere hasta n − 1 pasadas. Por cada pasada se comparan elementos adyacentes y se intercambian sus valores cuando el primer elemento es mayor que el segundo elemento. Al final de cada pasada, el elemento mayor ha «burbujeado» hasta la cima de la sublista actual. Por ejemplo, después que la pasada 0 está completa, la cola de la lista A[n − 1] está ordenada y el frente de la lista permanece desordenado. Las etapas del algoritmo son:

1. En la pasada 0 se comparan elementos adyacentes: (A[0],A[1]),(A[1],A[2]),(A[2],A[3]),...(A[n-2],A[n-1]) Se realizan n − 1 comparaciones, por cada pareja (A[i],A[i+1]) se intercambian los valores si A[i+1] < A[i]. Al final de la pasada, el elemento mayor de la lista está situado en A[n-1].

2. En la pasada 1 se realizan las mismas comparaciones e intercambios, terminando con el elemento segundo mayor valor en A[n-2].

3. El proceso termina con la pasada n − 1, en la que el elemento más pequeño se almacena en A[0].

Alternativa 4. Algoritmo de ordenación Shell:

Los pasos a seguir por el algoritmo para una lista de n elementos son:

1. Dividir la lista original en n/2 grupos de dos, considerando un incremento o salto entre los elementos de n/2.

2. Clarificar cada grupo por separado, comparando las parejas de elementos, y si no están ordenados, se intercambian.

3. Se divide ahora la lista en la mitad de grupos (n/4), con un incremento o salto entre los elementos también mitad (n/4), y nuevamente se clasifica cada grupo por separado.

4. Así sucesivamente, se sigue dividiendo la lista en la mitad de grupos que en el recorrido anterior con un incremento o salto decreciente en la mitad que el salto anterior, y luego clasificando cada grupo por separado.

5. El algoritmo termina cuando se consigue que el tamaño del salto es 1.

Alternativa 5. Algoritmo Quicksort:

La primera etapa en el algoritmo de partición es obtener el elemento pivote; una vez que se ha seleccionado se ha de buscar el sistema para situar en la sublista izquierda todos los elementos menores que el pivote y en la sublista derecha todos los elementos mayores que el pivote. Supongamos que todos los elementos de la lista son distintos, aunque será preciso tener en cuenta los casos en que existan elementos idénticos.

Los pasos que sigue el algoritmo quicksort:

1. Seleccionar el elemento central de a[0:n-1] como pivote
2. Dividir los elementos restantes en particiones izquierda y derecha, de modo que ningún elemento de la izquierda tenga una clave (valor) mayor que el pivote y que ningún elemento a la derecha tenga una clave más pequeña que la del pivote.
3. Ordenar la partición izquierda utilizando quicksort recursivamente.
4. Ordenar la partición derecha utilizando quicksort recursivamente.
5. La solución es partición izquierda seguida por el pivote y a continuación partición derecha.

Alternativa 6. Algoritmo de ordenación Binsort:

Se considera que el campo clave de los registros que se van a ordenar son números enteros en el rango 1 … m. Son necesarias m urnas por lo que es necesario definir un vector de m urnas. Las urnas pueden ser representadas por listas enlazadas, cada elemento de la lista contiene un registro cuyo campo clave es el correspondiente al de la urna en la que se encuentra. Así en la urna 1 se sitúan los registros cuyo campo clave sea igual a 1, en la urna 2 los registros cuyo campo clave sea 2, y así sucesivamente en la urna i se sitúan los registros cuyo campo clave sea igual a i. Una vez que se hayan distribuido los registros en las diversas urnas es necesario concatenar las listas. En la Figura 6.4 se muestra cómo realizar la concatenación.

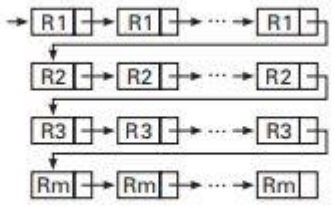


Figura 6.4. Concatenación de urnas representadas por listas enlazadas.

Alternativa 7. Algoritmo de ordenación Radixsort:

La idea clave de la ordenación Radixsort (también llamada por residuos) es clasificar por urnas primero respecto al dígito de menor peso (menos significativo) dk, después concatenar las urnas, clasificar de nuevo respecto al siguiente dígito dk−1, y así sucesivamente se sigue con el siguiente dígito hasta alcanzar el dígito más significativo dl , en ese momento la secuencia estará ordenada. La concatenación de las urnas consiste en enlazar el final de una con el frente de la siguiente. Al igual que en el método de Binsort, las urnas se representan mediante un vector de listas. En el caso de que la clave respecto a la que se ordena sea un entero, se tendrán 10 urnas, numeradas de 0 a 9. Si la clave respecto a la que se ordena es alfabética, habrá tantas urnas como letras distintas, desde la urna que represente a la letra a hasta la z. Para el caso de que clave sea entera, en primer lugar se determina el máximo número de dígitos que puede tener la clave. En un bucle de tantas iteraciones como máximo de dígitos se realizan las acciones de distribuir por

urnas los registros, concatenar... La distribución por urnas exige obtener el dígito del campo clave que se encuentra en la posición definida por el bucle externo, dicho dígito será el índice de la urna.

Paso 4. Transición de las Ideas a los Diseños Preliminares

En este paso descartaremos los algoritmos que no son factibles al mostrar el análisis de complejidad temporal de cada uno.

Algoritmo de selección:

Al algoritmo de ordenamiento por selección, para ordenar un vector de **n** términos, tiene que realizar siempre el mismo número de comparaciones:

$$c(n) = \frac{n^2 - n}{2}$$

Esto es, el número de comparaciones **c(n)** no depende del orden de los términos, si no del número de términos.

$$\Theta(c(n)) = n^2$$

Por lo tanto la cota ajustada asintótica del número de comparaciones pertenece al orden de n cuadrado.

El número de intercambios **i(n)**, también es fijo, téngase en cuenta que la instrucción:

intercambiar(lista[i], lista[mínimo])

siempre se ejecuta, aun cuando **i= mínimo**, lo que da lugar:

$$i(n) = n$$

sea cual sea el vector, y el orden de sus términos, lo que implica en todos los casos un coste lineal:

$$\Theta(i(n)) = n$$

la cota ajustada asintótica del número de intercambios es lineal, del orden de n.

Asimismo, la fórmula que representa el rendimiento del algoritmo, viene dada por la función:

$$c(n) = \frac{n^2 + n}{2}$$

https://es.wikipedia.org/wiki/Ordenamiento_por_selecci%C3%B3n

Algoritmo de inserción:

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

- Consideremos como barómetro la instrucción j>=0 y x<A[j]
- Para el valor "i", en el peor caso x = A[i] se tiene que comparar con los valores A[i-1], …, A[0] (i veces)
- Por tanto, el número total de operaciones a realizar es:

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} - 1$$

El orden máximo de este algoritmo esta en $O(n^2)$.

https://www.cs.buap.mx/~iolmos/ada/AlgoritmosOrdenamientoP1.pdf

Algoritmo de Burbuja:

```
void bubbleSort(int numbers[], int
    array_size)
{ int i, j, temp;
  for (i = (array_size - 1); i >= 0; i--)
  { for (j = 1; j <= i; j++)
      { if (numbers[j-1] > numbers[j])
        { temp = numbers[j-1];
          numbers[j-1] = numbers[j];
          numbers[j] = temp;
        }
      }
  }
}
```

i	j
n-1	n-1
n-2	n-2
n-3	n-3
…	…
0	0

$$total = \sum_{k=1}^{n-1} k = \sum_{k=1}^{n} k - n = \frac{n(n+1)}{2} - n$$

Claramente, el algoritmo esta en $O(n^2)$

Algoritmo de ordenación Shell:

Su implementación original, requiere O(n2) comparaciones e intercambios en el peor caso. Un cambio menor presentado en el libro de V. Pratt produce una implementación con un rendimiento de O(n log2 n) en el peor caso. Esto es mejor que las O(n2) comparaciones requeridas por algoritmos simples pero peor que el óptimo O(n log n). Aunque es fácil desarrollar un sentido intuitivo de cómo funciona este algoritmo, es muy difícil analizar su tiempo de ejecución.

https://es.wikipedia.org/wiki/Ordenamiento_Shell

Algoritmo Quicksort:

El peor caso: Si se elige como pivote el primer elemento del vector y además se considera que el vector esta ordenado decrecientemente entonces, el bucle '... Para cada elemento...' se ejecutará en total:

(n-1)+(n-2)+(n-3)+....+1

Cada miembro de este sumando proviene de cada una de las sucesivas ordenaciones recursivas. Este sumatorio da lugar a la siguiente expresión:

$$\sum_{i=1}^{n} (n - i) = \frac{[(n-1)+1](n-1)}{2} = \frac{n(n-1)}{2}$$

Que es de orden cuadrático T(n) ∈O(n 2)

Como se puede apreciar si la elección del pivote es tal que los subvectores son de igual tamaño y además los vectores están ordenados de forma aleatoria entonces el algoritmo posee una eficiencia del tipo O(n log n).

Recorrido del vector Numero de divisiones

https://www2.infor.uva.es/~jvalvarez/docencia/tema5.pdf

Algoritmo de ordenación Bin sort:

El **algoritmo del cartero** es una variante del bucket sort utilizada cuando los elementos a ordenar disponen de varias claves y/o subclaves. El nombre de este algoritmo viene del ejemplo de las oficinas postales; allí cuando hay que clasificar una carta para que llegue a su destino primero se clasifica según el país de destino, luego la ciudad o la región, después según la calle o el barrio de destino, etc. Es decir, este algoritmo utiliza varias claves para hacer ordenamientos sucesivos. La complejidad computacional es de O(cn), siendo c el número de claves que se utilizan para clasificar.

https://es.wikipedia.org/wiki/Ordenamiento_por_casilleros

Algoritmo de ordenación Radix sort:

La complejidad temporal del algoritmo es el siguiente: supongamos que los números de entrada n tiene dígitos máximo k. A continuación, el procedimiento se le llama ordenar contado con un total k veces. Counting sort es un algoritmo lineal o O(n). Así que todo el procedimiento de Radix

sort toma tiempo o (kn). Si los números son de tamaño finito, el algoritmo se ejecuta en O(n) tiempo asintótica.

Despues de mostrar la complejidad temporal de cada uno de los algoritmos de ordenamiento se presenta una tabla con el fin de resumir gráficamente cada una las eficiencias obtenidas. Se toma la decisión de descartar los métodos *Burbuja, Selección, Insercion, Radix.*

Nombre	Complejidad	Método
Burbuja	$O(n^2)$	Intercambio
Selección	$O(n^2)$	Selección
Inserción	$O(n^2)$ "en el peor de los casos"	Inserción
QuickSort	Promedio: O(n log n) Peor Caso: $O(n^2)$	Partición
Radix Sort	O(nk)	No comparativo
Bin Sort	O(n)	No comparativo
Shell Sort	$O(n^{1.25})$	Inserción

Se procede a dar las ventajas y desventajas de los algoritmos restantes con el fin de modelar y modificar las ideas prometedoras para construir el diseño factible.

Algoritmo QuickSort:

Ventajas:

- Requiere de pocos recursos en comparación a otros métodos de ordenamiento.
- En la mayoría de los casos, se requiere aproximadamente N log N operaciones.
- Ciclo interno es extremadamente corto.
- No se requiere de espacio adicional durante ejecución (in-place processing).

Desventajas:

- Se complica la implementación si la recursión no es posible.
- Peor caso, se requiere N2
- Un simple error en la implementación puede pasar sin detección, lo que provocaría un rendimiento pésimo.
- No es útil para aplicaciones de entrada dinámica, donde se requiere reordenar una lista de elementos con nuevos valores.
- Se pierde el orden relativo de elementos idénticos.

https://quicksortweb.wordpress.com/2017/10/07/ventajas-desventajas-y-aplicaciones/

Algoritmo Bin Sort:

Ventajas:

- No requiere memoria adicional.
- Mejor rendimiento que el método de inserción clásico.
- 5 veces mas rápido que Bubble sort
- 2 veces mas rápido que Insertion Sort

Desventajas:

- Implementación algo confusa.
- Realiza numerosas comparaciones e intercambios
- Shell Sort es significativamente mas lento que merge, heap y quick sort.

https://sites.google.com/site/algoritmoshellsort/ventajas-y-desventajas

Algoritmo Shell Sort:

Ventajas:
-No requiere memoria adicional.
-Mejor rendimiento que el método de inserción rápido.
-Fácil implantación.

Desventajas:
-Su funcionamiento puede resultar confuso
-Suele ser un poco lento.
-Realiza numerosas comparaciones e intercambios.

http://eodjesusv.blogspot.com/2013/11/metodo-shell-sort.html

Paso 5. Evaluación y Selección de la Mejor Solución

A continuación definiremos unos criterios con su respectiva numeración la cual nos permitirán evaluar las alternativas de la solución y con base en este resultado se elegirá la mejor alternativa que satisfaga las necesidades del problema planteado.

- Criterio A. Estabilidad del algoritmo. Se pretende que el algoritmo sea estable a la hora de ejecutarse:

  - ✓ [3] Estables
  - ✓ [2] Cuestionables o imprácticas
  - ✓ [1] Inestables

- Criterio B. Generalidad. Se busca definir qué tan general o especifico es el algoritmo:

  - ✓ [2] General
  - ✓ [1] Especifico

- Criterio C. Complejidad Espacial. Se busca definir que tanto espacio el algoritmo puede ocupar al resolver el problema.

    - ✓ [3] Poco
    - ✓ [2] Medio
    - ✓ [1] Mucho

	Criterio A	Criterio B	Criterio C	Total
Alternativa 1. Algoritmo QuickSort	Exacta 1	Exacta 2	Exacta 3	6
Alternativa 2. Algoritmo BinSort	Exacta 3	Exacta 1	Exacta 1	5
Alternativa 3. Algoritmo ShellSort	Exacta 1	Exacta 2	Exacta 2	5

Selección:

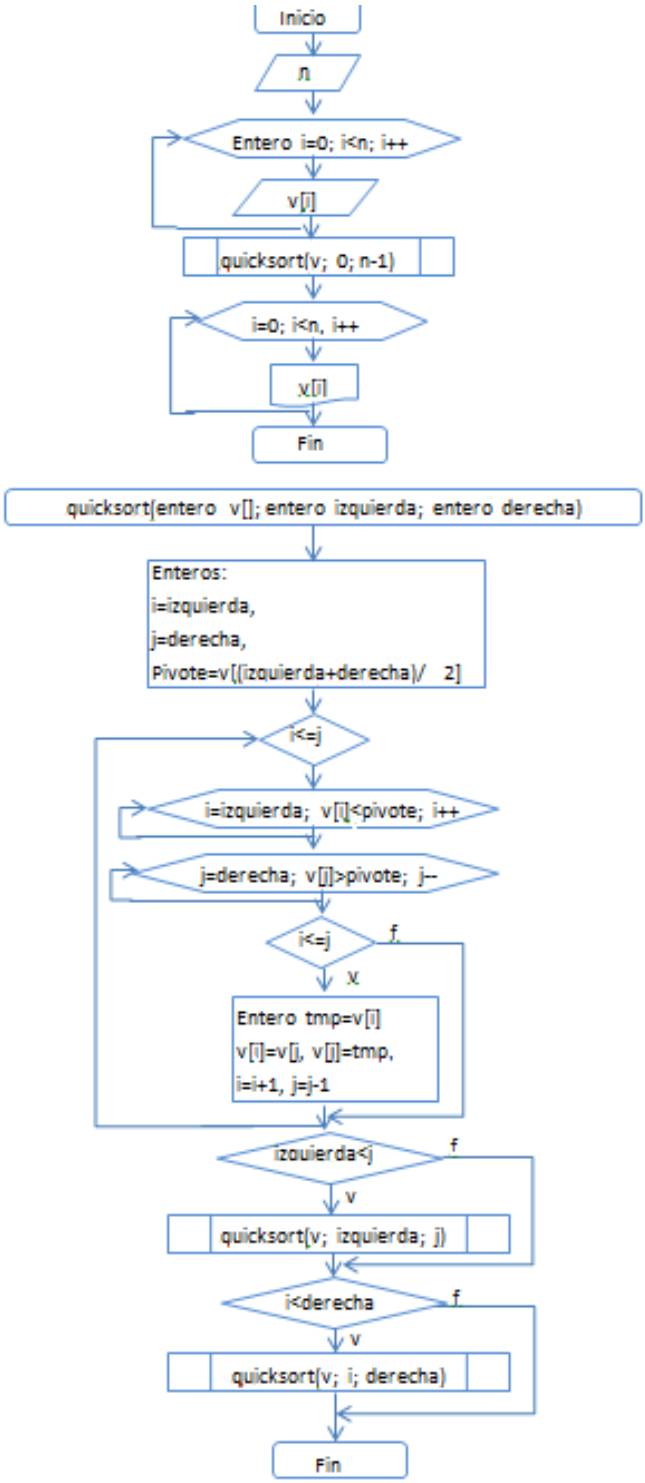
De acuerdo con la evaluación anterior se debe seleccionar la Alternativa 1 (Quicksort), ya que obtuvo la mayor puntuación de acuerdo con los criterios definidos.

Paso 6. Preparación de Informes y Especificaciones

Especificación del Problema (en términos de entrada/salida)

Problema: Organizar una cantidad de números determinados eficientemente.
Entradas: Cantidad determinada de números en desorden.
Salida: La misma cantidad de números esta vez organizada.

Diagrama de flujo del algoritmo



http://diagramadeflujodeordenamiento.blogspot.com/

Pseudocódigo del algoritmo

```
INICIO
Llenar(A)
Algoritmo
quicksort(A,inf,sup)
i<-inf
j<-sup
x<-A[(inf+sup)div 2]
mientras i=<j hacer
  mientras A[i]< x hacer
     i<-i+1
  fin_mientras
  mientras A[j]>x hacer
     j<- j-1
  fin_mientras
  si i=<j entonces
    tam<-A[i]
    A[i]<-A[j]
    A[j]<-tam
    i=i+1
    j=j-1
  fin_si
fin_mientras
si inf<j
   llamar_a
quicksort(A,inf,j)
fin_si
si i<sup
   llamar_a
quicksort(A,i,sup)
fin_si
FIN
```

https://quicksort.neocities.org/pseudocodigo.html