UNIVERSIDAD NACIONAL DE SAN AGUSTÍN DE AREQUIPA

ESCUELA PROFESIONAL DE CIENCIA DE LA COMPUTACIÓN

ALGORITMOS DE ORDENAMIENTO

Curso : Estructuras de Datos Avanzadas

Alumno

HERRERA COOPER MIGUEL ALEXANDER

Docente

MACHACA ARCEDA, VICENTE

Índice

1. Introduction

2. Conceptos Básicos

2.1. Algoritmo

Es un conjunto de instrucciones o reglas definidas y no-ambiguas, ordenadas y finitas que permite, típicamente, solucionar un problema, realizar un cómputo, procesar datos y llevar a cabo otras tareas o actividades.

2.2. Lenguaje de Programación

Un lenguaje de programación es un lenguaje formal que proporciona una serie de instrucciones que permiten a un programador escribir secuencias de órdenes y algoritmos a modo de controlar el comportamiento físico y lógico de una computadora con el objetivo de que produzca diversas clases de datos. Ejemplos:

- Python
- C++
- Java

Son los lenguajes que usaremos para nuestros algoritmos.

2.3. Complejidad Computacional

- Estudia el orden de complejidad de un algoritmo que resuelve un problema decidible.
- Para ello, considera los 2 tipos de recursos requeridos durante el cómputo para resolver un problema:
  - **Tiempo**: Número de pasos base de ejecución de un algoritmo para resolver un problema.
  - **Espacio**: Cantidad de memoria utilizada para resolver un problema.
- La complejidad de un algoritmo se expresa como función del tamaño de la entrada del problema, n.
- Se refiere al ratio de crecimiento de los recursos con respecto a n:
  - Ratio del Tiempo de ejecución (Temporal): T(n).
  - Ratio del Espacio de almacenamiento necesario (Espacial): S(n).

3. Algoritmos de Ordenamiento

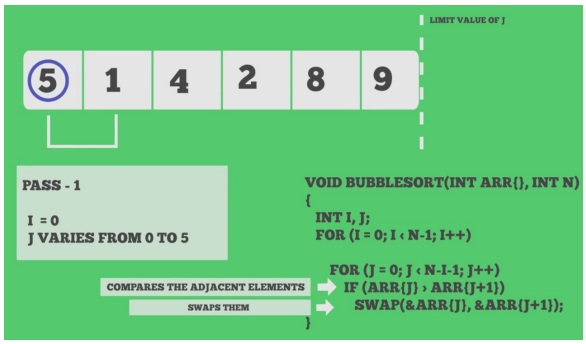
3.1. Bubble Sort

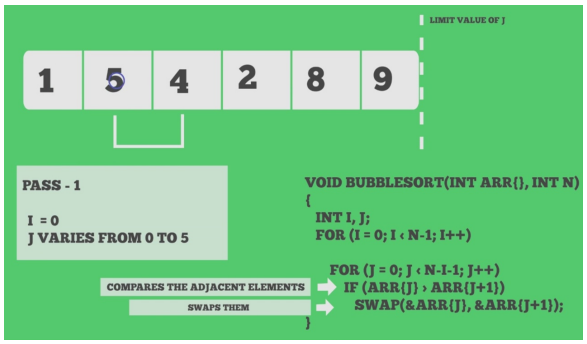
- Bubble Sort es el algoritmo de clasificación más simple que funciona intercambiando repetidamente los elementos adyacentes si están en orden incorrecto.
- Debido a su simplicidad, la clasificación de burbujas se usa a menudo para introducir el concepto de un algoritmo de clasificación.
- **Complejidad**:
  - **Mejor Caso**: O(n) - Matriz ordenada.
  - **Caso Promedio**: O(n).
  - **Peor Caso**:O(n) - Matriz ordenada inversamente.
- **Espacio Auxiliar**: O(1).
- **Estable**: Sí.
- **Algoritmo**:

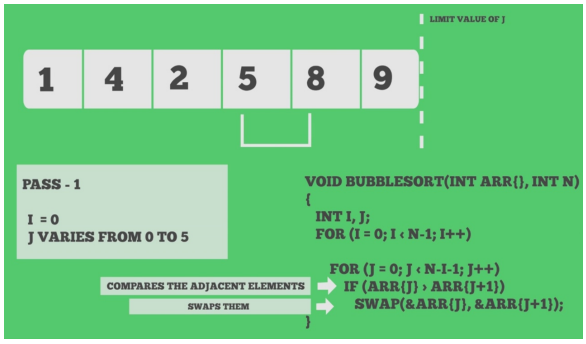
```
begin BubbleSort(list)

   for all elements of list
      if list[i] > list[i+1]
         swap(list[i], list[i+1])
      end if
   end for

   return list

end BubbleSort
```

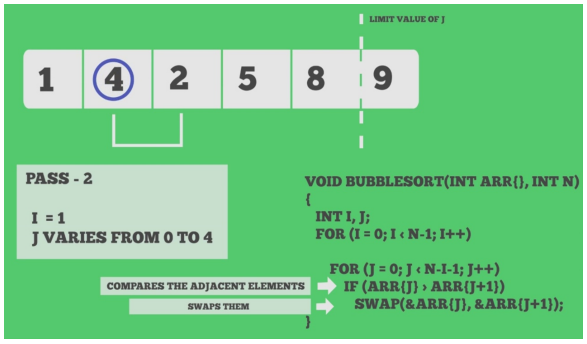
- **Usos**:
  - En los gráficos de computadora es popular por su capacidad de detectar un error muy pequeño (como el intercambio de solo dos elementos) en arreglos casi ordenados y arreglarlo con una complejidad lineal (2n).
  - Por ejemplo, se usa en un algoritmo de relleno de polígonos, donde las líneas de delimitación se ordenan por su coordenada x en una línea de exploración específica (una línea paralela al eje x) y al aumentar y su orden cambia (se intercambian dos elementos) solo en las intersecciones de dos líneas.

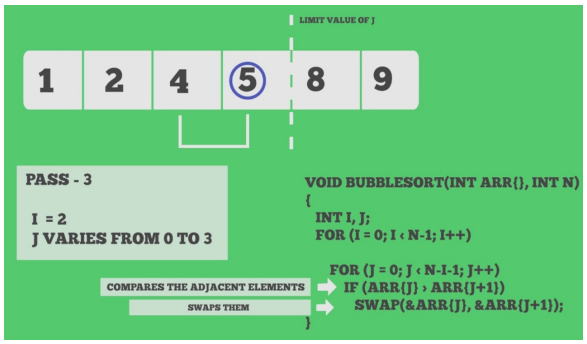
- **Ejemplo**:

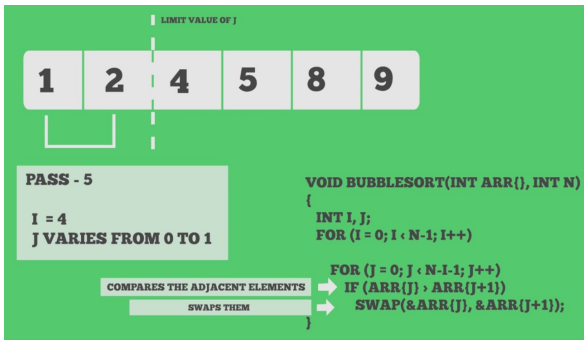

LIMIT VALUE OF J
5 1 4 2 8 9
PASS - 1
I = 0
J VARIES FROM 0 TO 5
VOID BUBBLESORT(INT ARR{}, INT N)
{
INT I, J;
FOR (I = 0; I < N-1; I++)
FOR (J = 0; J < N-I-1; J++)
COMPARES THE ADJACENT ELEMENTS
IF (ARR{J} > ARR{J+1})
SWAPS THEM
SWAP(&ARR{J}, &ARR{J+1});
}


LIMIT VALUE OF J
1 5 4 2 8 9
PASS - 1
I = 0
J VARIES FROM 0 TO 5
VOID BUBBLESORT(INT ARR{}, INT N)
{
INT I, J;
FOR (I = 0; I < N-1; I++)
FOR (J = 0; J < N-I-1; J++)
COMPARES THE ADJACENT ELEMENTS
IF (ARR{J} > ARR{J+1})
SWAPS THEM
SWAP(&ARR{J}, &ARR{J+1});
}


LIMIT VALUE OF J
1 4 2 5 8 9
PASS - 1
I = 0
J VARIES FROM 0 TO 5
VOID BUBBLESORT(INT ARR{}, INT N)
{
INT I, J;
FOR (I = 0; I < N-1; I++)
FOR (J = 0; J < N-I-1; J++)
COMPARES THE ADJACENT ELEMENTS
IF (ARR{J} > ARR{J+1})
SWAPS THEM
SWAP(&ARR{J}, &ARR{J+1});
}


LIMIT VALUE OF J
1 4 2 5 8 9
PASS - 2
I = 1
J VARIES FROM 0 TO 4
VOID BUBBLESORT(INT ARR{}, INT N)
{
INT I, J;
FOR (I = 0; I < N-1; I++)
FOR (J = 0; J < N-I-1; J++)
COMPARES THE ADJACENT ELEMENTS
IF (ARR{J} > ARR{J+1})
SWAPS THEM
SWAP(&ARR{J}, &ARR{J+1});
}


LIMIT VALUE OF J
1 2 4 5 8 9
PASS - 3
I = 2
J VARIES FROM 0 TO 3
VOID BUBBLESORT(INT ARR{}, INT N)
{
INT I, J;
FOR (I = 0; I < N-1; I++)
FOR (J = 0; J < N-I-1; J++)
COMPARES THE ADJACENT ELEMENTS
IF (ARR{J} > ARR{J+1})
SWAPS THEM
SWAP(&ARR{J}, &ARR{J+1});
}


LIMIT VALUE OF J
1 2 4 5 8 9
PASS - 5
I = 4
J VARIES FROM 0 TO 1
VOID BUBBLESORT(INT ARR{}, INT N)
{
INT I, J;
FOR (I = 0; I < N-1; I++)
FOR (J = 0; J < N-I-1; J++)
COMPARES THE ADJACENT ELEMENTS
IF (ARR{J} > ARR{J+1})
SWAPS THEM
SWAP(&ARR{J}, &ARR{J+1});
}

- **Implementación en Python**

```python
def bubbleSort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(0, n-i-1):
            if arr[j] > arr[j+1] :
                arr[j], arr[j+1] = arr[j+1], arr[j]
```

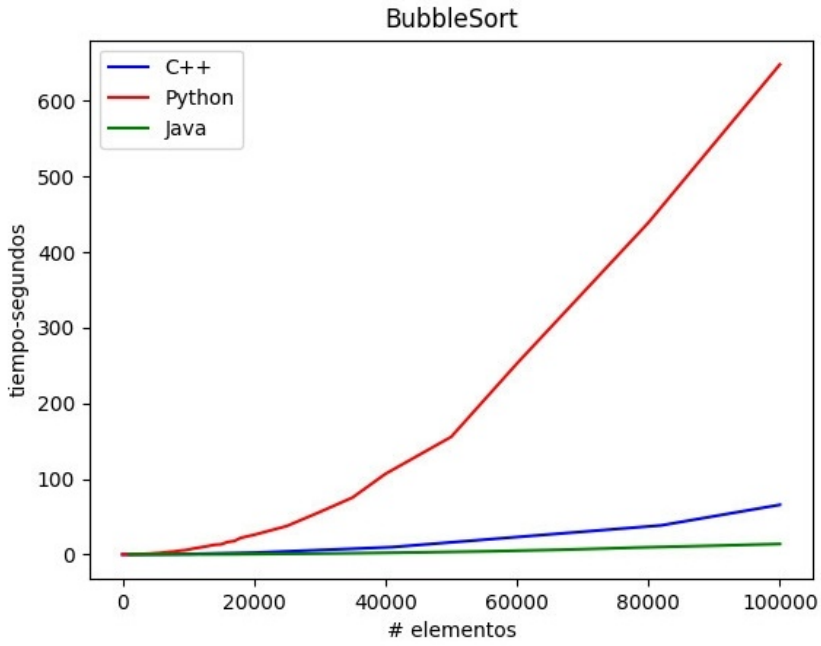
- **Implementación en C++**

```cpp
void swap(int &a, int &b){
    int temp = a;
    a = b;
    b = temp;
}

void bubbleSort(vector<int>&a){
  int tamanho = a.size();
  bool swapped;
  for (int i = 0; i < tamanho-1; i++){
    swapped = false;
    for (int j = 0; j < tamanho-i-1; j++){
      if (a[j] > a[j+1]){
          swap(a[j], a[j+1]);
          swapped = true;
      }
    }

    if(!swapped) break;
  }
}
```

- **Implementación en Java**

```java
class BubbleSort
{
    void bubbleSort(int arr[])
    {
        int n = arr.length;
        for (int i = 0; i < n-1; i++)
            for (int j = 0; j < n-i-1; j++)
                if (arr[j] > arr[j+1])
                {
                    int temp = arr[j];
                    arr[j] = arr[j+1];
                    arr[j+1] = temp;
                }
    }


    void printArray(int arr[])
    {
        int n = arr.length;
        for (int i=0; i<n; ++i)
            System.out.print(arr[i] + " ");
        System.out.println();
    }

    public static void main(String args[])
    {
        int n=10000;
        int arr[]=new int[n];
        for(int i=0;i<n;i++)
                arr[i]=(int)(Math.random()*100000)+1;
        BubbleSort ob = new BubbleSort();
        ob.bubbleSort(arr);
        System.out.println("Sorted array");
        ob.printArray(arr);
    }
}
```

- **Comparacion del Algoritmo en los 3 lenguajes**



Se puede apreciar en la imagen que Bubble sort presenta un comportamiento más rápido al ser implementado en Java, que al ser implementado en Python requiere más tiempo para su ejecución en el procesamiento de los datos.

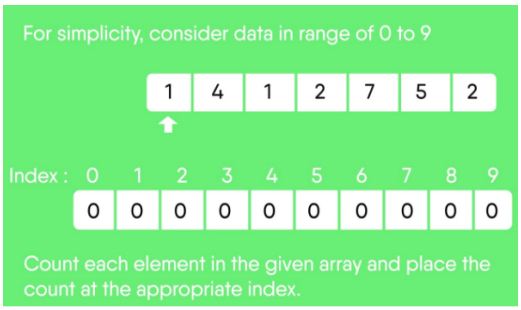
3.2. Counting Sort

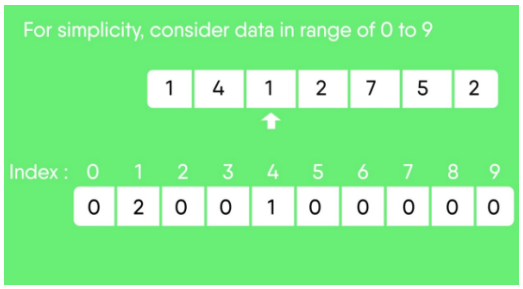
- Es una técnica de clasificación basada en claves entre un rango específico.
- Funciona contando el número de objetos que tienen valores clave distintos (tipo de hash).
- Luego haciendo algo de aritmética para calcular la posición de cada objeto en la secuencia de salida.
- **Complejidad**: O(n+k)
  n es el número de elementos en la matriz.
  k es el rango de entrada.
- **Espacio Auxiliar**: O(n+k)
- **Estable**: Sí
- **Algoritmo**

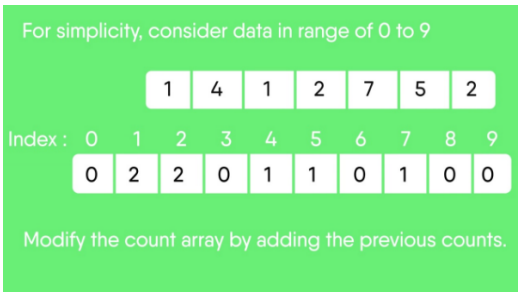
```
Begin
   max = get maximum element from array.
   define count array of size [max+1]

   for i := 0 to max do
      count[i] = 0 //set all elements in the count array to 0
   done

   for i := 1 to size do
      increase count of each number which have found in the array
   done

   for i := 1 to max do
      count[i] = count[i] + count[i+1] //find cumulative frequency
   done

   for i := size to 1 decrease by 1 do
      store the number in the output array
      decrease count[i]
   done

   return the output array
End
```

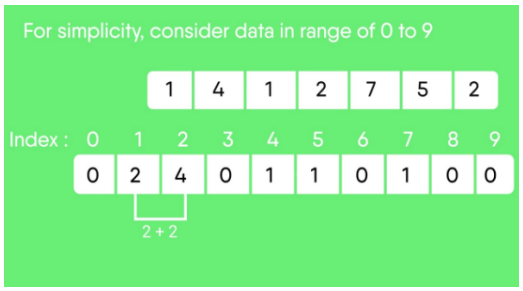
- **Usos**:
  Sólo puede ser utilizado por tanto para ordenar elementos que sean contables (como los números enteros en un determinado intervalo, pero no los números reales, por ejemplo).

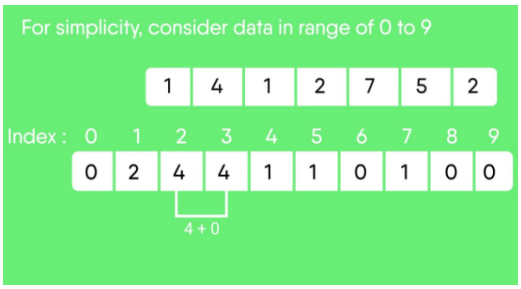
- **Ejemplo**:

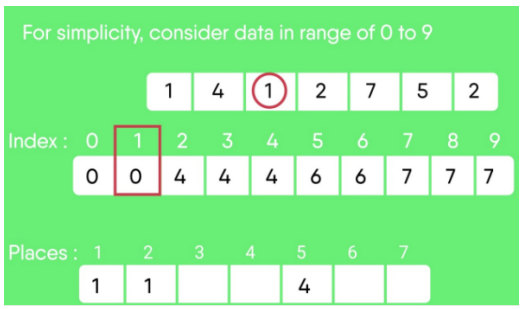

For simplicity, consider data in range of 0 to 9
1 4 1 2 7 5 2
Index : 0 1 2 3 4 5 6 7 8 9
0 0 0 0 0 0 0 0 0 0
Count each element in the given array and place the count at the appropriate index.


For simplicity, consider data in range of 0 to 9
1 4 1 2 7 5 2
Index : 0 1 2 3 4 5 6 7 8 9
0 2 0 0 1 0 0 0 0 0


For simplicity, consider data in range of 0 to 9
1 4 1 2 7 5 2
Index : 0 1 2 3 4 5 6 7 8 9
0 2 2 0 1 1 0 1 0 0
Modify the count array by adding the previous counts.


For simplicity, consider data in range of 0 to 9
1 4 1 2 7 5 2
Index : 0 1 2 3 4 5 6 7 8 9
0 2 4 0 1 1 0 1 0 0
2 + 2


For simplicity, consider data in range of 0 to 9
1 4 1 2 7 5 2
Index : 0 1 2 3 4 5 6 7 8 9
0 2 4 4 1 1 0 1 0 0
4 + 0


For simplicity, consider data in range of 0 to 9
1 4 1 2 7 5 2
Index : 0 1 2 3 4 5 6 7 8 9
0 0 4 4 4 6 6 7 7 7
Places : 1 2 3 4 5 6 7
1 1 4


For simplicity, consider data in range of 0 to 9
1 4 1 2 7 5 2
Index : 0 1 2 3 4 5 6 7 8 9
0 0 2 4 4 5 6 6 7 7
Places : 1 2 3 4 5 6 7
1 1 2 2 4 5 7

- **Implementación en Python**

```python
def countingSort(tlist, k):
    count_list = [0]*(k)
    for n in tlist:
        count_list[n] = count_list[n] + 1
    i=0
    for n in range(len(count_list)):
        while count_list[n] > 0:
            tlist[i] = n
            i+=1
            count_list[n] -= 1

def countSort(arr):
    countingSort(arr, max(arr)+1)
```

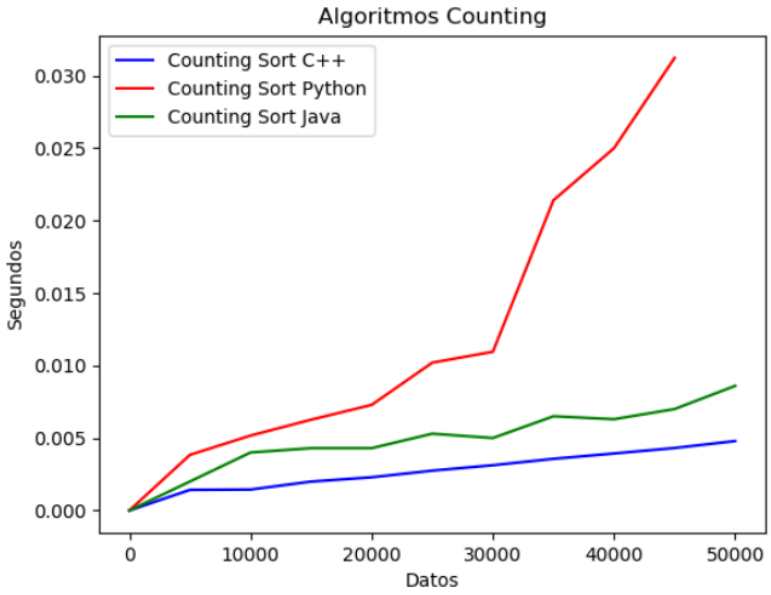
- **Implementación en C++**

```cpp
void countSort(vector <int>& arr)
{
    int max = *max_element(arr.begin(), arr.end());
    int min = *min_element(arr.begin(), arr.end());
    int range = max - min + 1;

    vector<int> count(range), output(arr.size());
    for(int i = 0; i < arr.size(); i++)
        count[arr[i]-min]++;

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

    for(int i = arr.size()-1; i >= 0; i--){
         output[ count[arr[i]-min] -1 ] = arr[i];
              count[arr[i]-min]--;
    }

    for(int i=0; i < arr.size(); i++)
            arr[i] = output[i];
}
```

- **Implementación en Java**

```java
class CountingSort
{

    static void countSort(int[] arr)
    {
        int max = Arrays.stream(arr).max().getAsInt();
        int min = Arrays.stream(arr).min().getAsInt();
        int range = max - min + 1;
        int count[] = new int[range];
        int output[] = new int[arr.length];
        for (int i = 0; i < arr.length; i++)
        {
            count[arr[i] - min]++;
        }

        for (int i = 1; i < count.length; i++)
        {
            count[i] += count[i - 1];
        }

        for (int i = arr.length - 1; i >= 0; i--)
        {
            output[count[arr[i] - min] - 1] = arr[i];
            count[arr[i] - min]--;
        }

        for (int i = 0; i < arr.length; i++)
        {
            arr[i] = output[i];
        }
    }
```

- **Comparación del Algoritmo en los 3 lenguajes** Counting Sort no tiene mucho tiempo de demora pero ocupa mucha cantidad de espacio, sin embargo Python sigue siendo muy lento y al igual que los demás algoritmos Java y C++ tratan de tener una mejor performance.



El ordenamiento de conteo es eficiente si el rango de datos de entrada no es significativamente mayor que el número de objetos a ordenar. Considere la situación en la que la secuencia de entrada está entre el rango 1 a 10K y los datos son 10, 5, 10K, 5K.

3.3. HeapSort

- Heap Sort es una técnica de clasificación basada en la comparación basada en la estructura de datos del montón binario.
- Es similar al orden de selección donde primero encontramos el elemento máximo y colocamos el elemento máximo al final. Repetimos el mismo proceso para el elemento restante.
- **¿Qué es el Montículo Binario ?**

  Un montón binario es un árbol binario completo donde los elementos se almacenan en un orden especial, de modo que el valor en un nodo principal es mayor (o menor) que los valores en sus dos nodos secundarios.

  El primero se llama montón máximo y el segundo se llama montón mínimo.
  El montón se puede representar por árbol binario o matriz.

- **¿Por qué la representación basada en matriz para Binary Heap?**

  Dado que un montón binario es un árbol binario completo, se puede representar fácilmente como matriz y la representación basada en matriz es eficiente en el espacio.

  Si el nodo padre se almacena en el índice I, el hijo izquierdo se puede calcular con 2 * I + 1 y el hijo derecho con 2 * I + 2 (suponiendo que la indexación comienza en 0).

- **Algoritmo de clasificación del Montículo para ordenar de forma creciente:**

  - Cree un montón máximo a partir de los datos de entrada.
  - En este punto, el elemento más grande se almacena en la raíz del montón. Reemplácelo con el último elemento del montón seguido de reducir el tamaño del montón en 1. Finalmente, heapifique la raíz del árbol.
  - Repita los pasos anteriores mientras el tamaño del montón es mayor que 1.

- **¿Cómo construir el Montículo?**
  El procedimiento Heapify se puede aplicar a un nodo solo si sus nodos hijos están heapified. Por lo tanto, la heapificación debe realizarse en el orden ascendente.

```
Datos de entrada: 4, 10, 3, 5, 1
         4 (0)
        / \
     10 (1) 3 (2)
    / \
 5 (3) 1 (4)

Los números entre paréntesis representan los índices en la matriz.
representación de datos.

Aplicando el procedimiento heapify al índice 1:
         4 (0)
        / \
    10 (1) 3 (2)
    / \
5 (3) 1 (4)

Aplicando el procedimiento heapify al índice 0:
        10 (0)
        / \
     5 (1) 3 (2)
    / \
 4 (3) 1 (4)
```

  El procedimiento de heapify se llama a sí mismo de forma recursiva para construir heap de arriba hacia abajo.

- **Complejidad**:
  Heap : O(logn)
  BuildHeap : O(n)
  Heap Sort : O(n logn)

- **Espacio Auxiliar**:O(n logn)

- **Estable**: No

- **Algortimo**

```
Heapsort(A) {
   BuildHeap(A)
   for i <- length(A) downto 2 {
      exchange A[1] <-> A[i]
      heapsize <- heapsize -1
      Heapify(A, 1)
}
```

```
BuildHeap(A) {
   heapsize <- length(A)
   for i <- floor( length/2 ) downto 1
      Heapify(A, i)
}
```

```
Heapify(A, i) {
   le <- left(i)
   ri <- right(i)
   if (le<=heapsize) and (A[le]>A[i])
      largest <- le
   else
      largest <- i
   if (ri<=heapsize) and (A[ri]>A[largest])
      largest <- ri
   if (largest != i) {
      exchange A[i] <-> A[largest]
      Heapify(A, largest)
   }
}
```

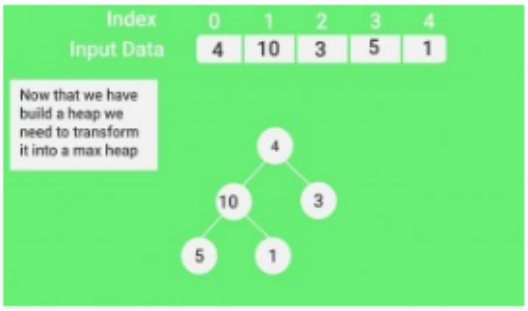
- **Usos**

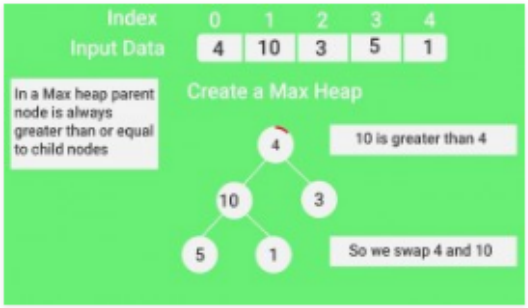
  1. Ordenar una matriz casi ordenada (o K ordenada)
  2. k elementos más grandes (o más pequeños) en una matriz

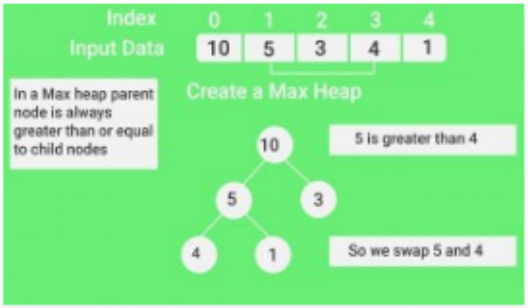
  El algoritmo de ordenamiento dinámico tiene usos limitados porque Quicksort y Mergesort son mejores en la práctica. Sin embargo, la estructura de datos Heap sí se usa enormemente.

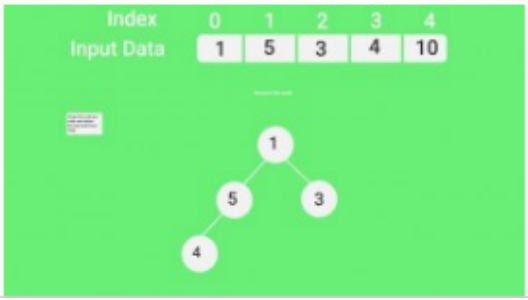
- **Ejemplo**

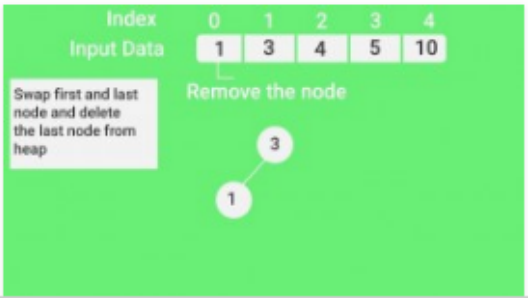

Index 0 1 2 3 4
Input Data 4 10 3 5 1
Build Heap
4


Index 0 1 2 3 4
Input Data 4 10 3 5 1
Now that we have build a heap we need to transform it into a max heap
4
10 3
5 1


Index 0 1 2 3 4
Input Data 4 10 3 5 1
In a Max heap parent node is always greater than or equal to child nodes
Create a Max Heap
10 is greater than 4
So we swap 4 and 10


Index 0 1 2 3 4
Input Data 10 5 3 4 1
In a Max heap parent node is always greater than or equal to child nodes
Create a Max Heap
5 is greater than 4
So we swap 5 and 4


Index 0 1 2 3 4
Input Data 1 5 3 4 10
1
5 3
4


Index 0 1 2 3 4
Input Data 1 3 4 5 10
Swap first and last node and delete the last node from heap
Remove the node
3
1

- **Implementacion del Algoritmo en Python**

```
def heapify(arr, n, i):
    largest = i
    l = 2 * i + 1
    r = 2 * i + 2
    if l < n and arr[i] < arr[l]:
        largest = l
    if r < n and arr[largest] < arr[r]:
        largest = r
    if largest != i:
        arr[i],arr[largest] = arr[largest],arr[i]
        heapify(arr, n, largest)

def heapSort(arr):
    n = len(arr)
    for i in range(n, -1, -1):
        heapify(arr, n, i)
    for i in range(n-1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]
        heapify(arr, i, 0)
```

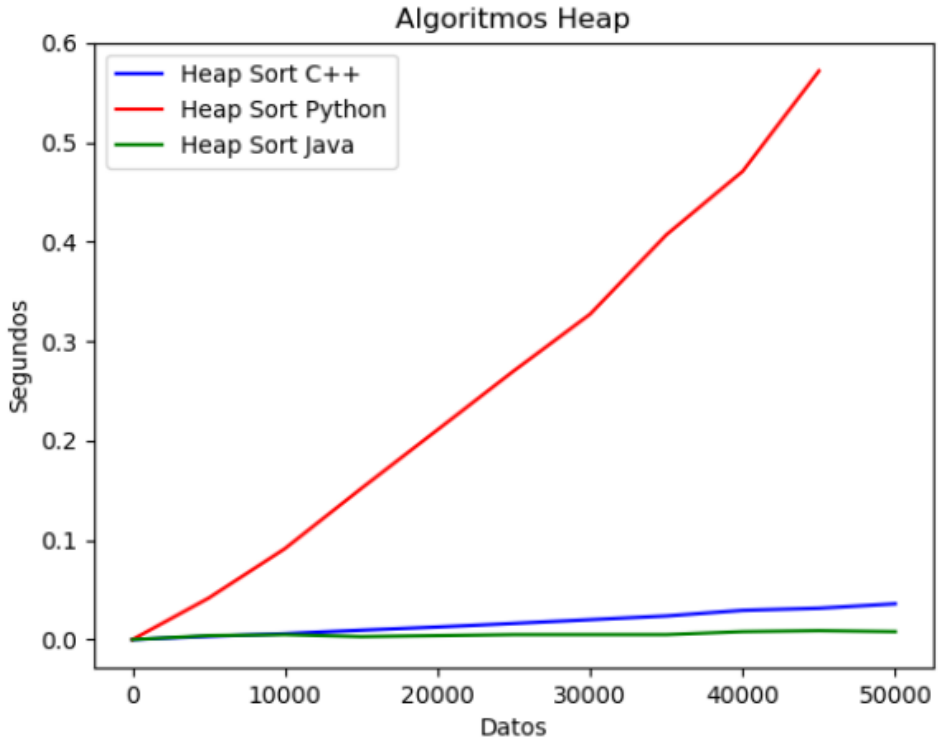
- **Implementación del Algoritmo en C++**

```
void heapify(vector<int>&arr, int n, int i)
{
    int largest = i;
    int l = 2*i + 1;
    int r = 2*i + 2;


    if (l < n && arr[l] > arr[largest])
        largest = l;

    if (r < n && arr[r] > arr[largest])
        largest = r;

    if (largest != i)
    {
        swap(arr[i], arr[largest]);
        heapify(arr, n, largest);
    }
}

void heapSort(vector<int>&arr, int n)
{
    for (int i = n / 2 - 1; i >= 0; i--)
antalla heapify(arr, n, i);

    for (int i=n-1; i>=0; i--)
    {
        swap(arr[0], arr[i]);
        heapify(arr, i, 0);
    }
}
```

- **Implementación del Algoritmo en Java**

```java
public class HeapSort
{
    public void sort(int arr[])
    {
        int n = arr.length;
        for (int i = n / 2 - 1; i >= 0; i--)
            heapify(arr, n, i);

        for (int i=n-1; i>=0; i--)
        {
            int temp = arr[0];
            arr[0] = arr[i];
            arr[i] = temp;
            heapify(arr, i, 0);
        }
    }

    void heapify(int arr[], int n, int i)
    {
        int largest = i;
        int l = 2*i + 1;
        int r = 2*i + 2;

        if (l < n && arr[l] > arr[largest])
            largest = l;

        if (r < n && arr[r] > arr[largest])
            largest = r;

        if (largest != i)
        {
            int swap = arr[i];
            arr[i] = arr[largest];
            arr[largest] = swap;
            heapify(arr, n, largest);
        }
    }
```

- **Comaparación del Algoritmo en los 3 lenguajes**



Heap Sort no memoria adicional y funciones también.
Con grandes cantidades de datos el algoritmo mejora pero como cualquier algoritmo hace uso de memoria y nuevamente Python es el más lento.
Su desempeño es en promedio tan bueno como el Quick Sort.

3.4. Insertion Sort

- La ordenación por inserción es un algoritmo de ordenación simple que funciona de la forma en que ordenamos las cartas en nuestras manos.
- **Complejidad**: O(n)
- **Espacio Auxiliar**: O(1)
- **Estable**: Sí
- **Algoritmo:**

```
procedure insertionSort( A : array of items )
   int holePosition
   int valueToInsert

   for i = 1 to length(A) inclusive do:

      /* select value to be inserted */
      valueToInsert = A[i]
      holePosition = i

      /*locate hole position for the element to be inserted */

      while holePosition > 0 and A[holePosition-1] > valueToInsert do:
         A[holePosition] = A[holePosition-1]
         holePosition = holePosition -1
      end while

      /* insert the number at hole position */
      A[holePosition] = valueToInsert

   end for

end procedure
```

- **Usos:**
  - La ordenación por inserción se utiliza cuando el número de elementos es pequeño.
  - También puede ser útil cuando la matriz de entrada está casi ordenada, solo unos pocos elementos están mal ubicados en una gran matriz completa.

- **Ejemplo**


4 3 2 10 12 1 5 6
4 3 2 10 12 1 5 6
3 4 2 10 12 1 5 6
2 3 4 10 12 1 5 6
2 3 4 10 12 1 5 6
2 3 4 10 12 1 5 6
1 2 3 4 10 12 5 6
1 2 3 4 5 10 12 6
1 2 3 4 5 6 10 12

- **Implementación en Python**

```python
def insertionSort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i-1
        while j >= 0 and key < arr[j] :
                arr[j + 1] = arr[j]
                j -= 1
        arr[j + 1] = key
```

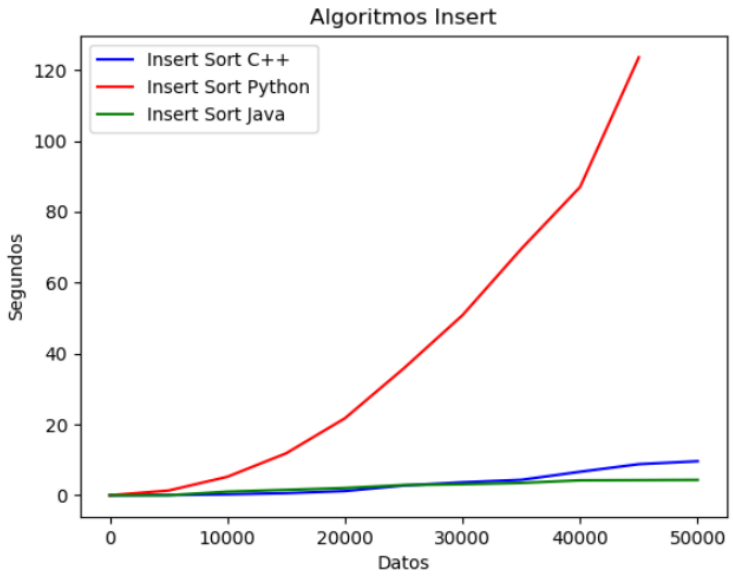
- **Implementación en C++**

```cpp
void insertionSort(vector<int>&arr)
{
    int key, j;
    for (int i = 1; i < arr.size(); i++)
    {
        key = arr[i];
        j = i - 1;
        while (j >= 0 && arr[j] > key)
        {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}
```

- **Implementación en Java**

```java
class InsertionSort {
    void sort(int arr[])
    {
        int n = arr.length;
        for (int i = 1; i < n; ++i) {
            int key = arr[i];
            int j = i - 1;
            while (j >= 0 && arr[j] > key) {
                arr[j + 1] = arr[j];
                j = j - 1;
            }
            arr[j + 1] = key;
        }
    }
```

- **Rendimiento del Algoritmo en los 3 Lenguajes**



A grandes cantidades de datos este algorimto tiene un buen rendimiento siendo implementado en C++ o Java, pero lamentablemente se puede apreciar que no sucede lo mismo con Python.

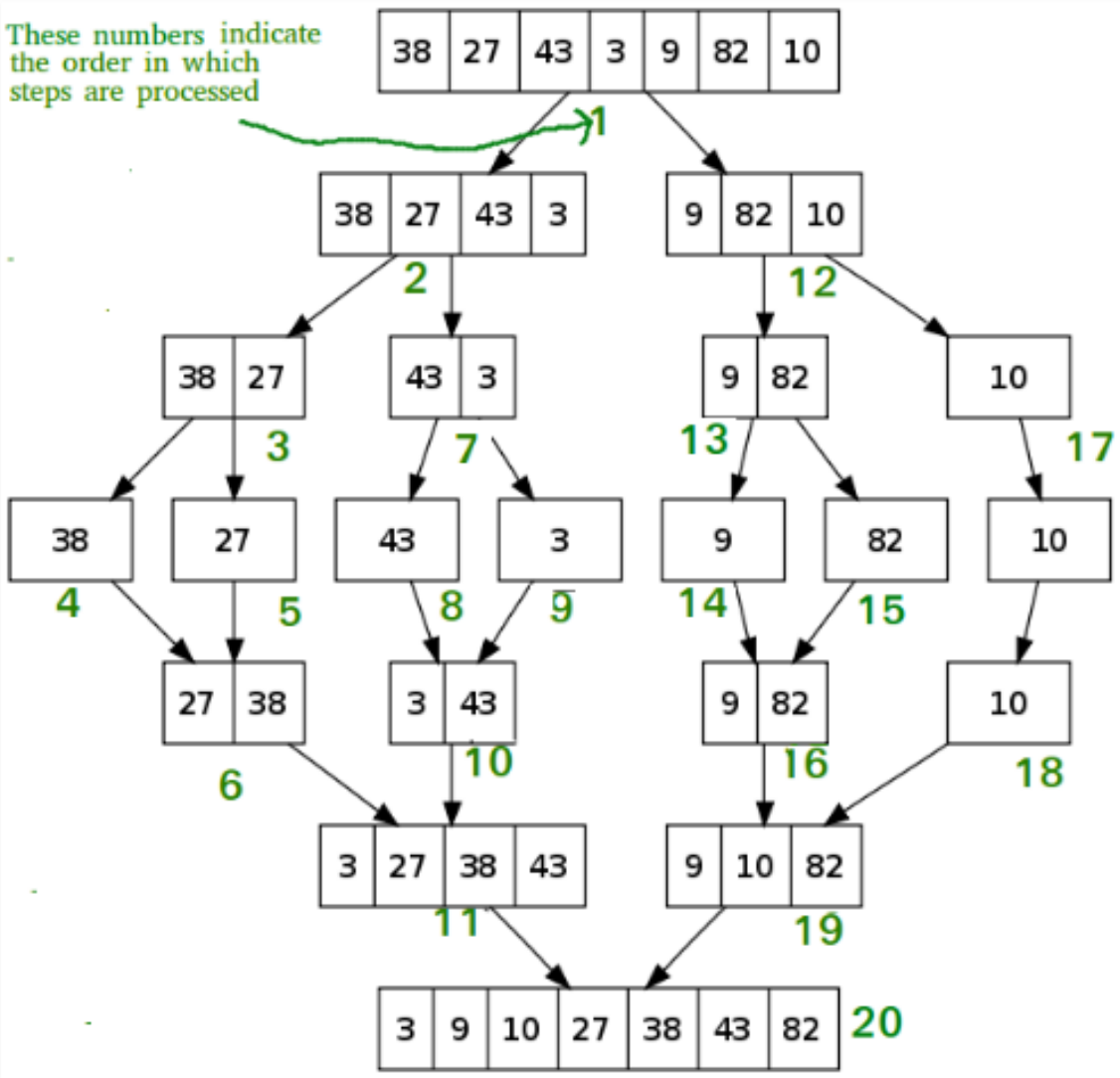
3.5. Merge Sort

- Al igual que QuickSort , Merge Sort es un algoritmo Divide and Conquer .
- Divide la matriz de entrada en dos mitades, se llama a sí misma para las dos mitades y luego combina las dos mitades ordenadas.
- La función merge () se usa para fusionar dos mitades. La combinación (arr, l, m, r) es un proceso clave que supone que arr [l..m] y arr [m + 1..r] se ordenan y fusiona las dos sub-matrices ordenadas en una.
- **Complejidad:**O(n log n)
- **Espacio Auxiliar:** O(n)
- **Estable:** Sí
- **Algoritmo:**

```
MergeSort (arr [], l, r)
Si r> l
     1. Encuentre el punto medio para dividir la matriz en dos mitades:
             medio m = (l + r) / 2
    2. Llame a mergeSort para la primera mitad:
             Llamar a mergeSort (arr, l, m)
     3. Llame a mergeSort para la segunda mitad:
             Llamar a mergeSort (arr, m + 1, r)
     4. Combine las dos mitades ordenadas en los pasos 2 y 3:
             Fusionar llamada (arr, l, m, r)
```

- **Usos:**
  - Problema de recuento de inversiones
  - Utilizado en clasificación externa

- **Ejemplo:**


These numbers indicate the order in which steps are processed
38 27 43 3 9 82 10
1
38 27 43 3
9 82 10
2
12
38 27
43 3
9 82
10
3
7
13
17
38
27
43
3
9
82
10
4
5
8
9
14
15
27 38
3 43
9 82
10
6
10
16
18
3 27 38 43
9 10 82
11
19
3 9 10 27 38 43 82
20

- **Implementación en Python**

```
def mergeSort(arr):
    if len(arr) >1:
        mid = len(arr)//2
        L = arr[:mid]
        R = arr[mid:]

        mergeSort(L)
        mergeSort(R)

        i = j = k = 0

        while i < len(L) and j < len(R):
            if L[i] < R[j]:
                arr[k] = L[i]
                i+=1
            else:
                arr[k] = R[j]
                j+=1
            k+=1

        while i < len(L):
            arr[k] = L[i]
            i+=1
            k+=1

        while j < len(R):
            arr[k] = R[j]
            j+=1
            k+=1
```

- **Implementación en C++**

```
void merge(vector<int>&arr, int l, int m, int r){
    int i, j, k;
    int n1 = m - l + 1;
    int n2 =  r - m;
    int L[n1], R[n2];

    for (i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[m + 1+ j];

    i = 0;
    j = 0;
    k = l;
    while (i < n1 && j < n2){
        if (L[i] <= R[j]){
            arr[k] = L[i];
            i++;
        }
        else{
            arr[k] = R[j];
            j++;
        }
        k++;
    }
    while (i < n1){
        arr[k] = L[i];
        i++;
        k++;
    }
    while (j < n2){
        arr[k] = R[j];
        j++;
        k++;
    }
}
```

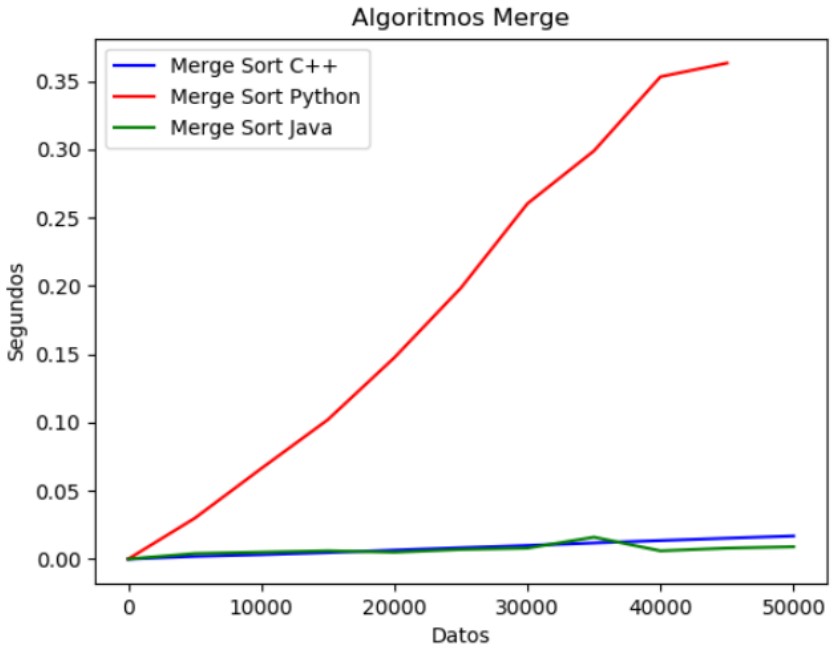
```
void mergeSort(vector<int>&arr, int l, int r){
    if (l < r){
        int m = l+(r-l)/2;
        mergeSort(arr, l, m);
        mergeSort(arr, m+1, r);
        merge(arr, l, m, r);
    }
}
```

- **Implementación en Java**

```
class MergeSort
{
    void merge(int arr[], int l, int m, int r) {
        int n1 = m - l + 1;
        int n2 = r - m;
        int L[] = new int [n1];
        int R[] = new int [n2];

        for (int i=0; i<n1; ++i)
            L[i] = arr[l + i];
        for (int j=0; j<n2; ++j)
            R[j] = arr[m + 1+ j];
        int i = 0, j = 0;
        int k = l;
        while (i < n1 && j < n2) {
            if (L[i] <= R[j]) {
                arr[k] = L[i];
                i++;
            }
            else{
                arr[k] = R[j];
                j++;
            }
            k++;
        }
        while (i < n1) {
            arr[k] = L[i];
            i++;
            k++;
        }
        while (j < n2) {
            arr[k] = R[j];
            j++;
            k++;
        }
    }
```

```
void sort(int arr[], int l, int r)
{
    if (l < r)
    {
        int m = (l+r)/2;
        sort(arr, l, m);
        sort(arr , m+1, r);
        merge(arr, l, m, r);
    }
}
```

- **Rendimiento del Algoritmo en los 3 lenguajes**.



Al igual que el Counting Sort tiene un gran eficiencia con respecto al tiempo pero también hace mucho uso de espacio debido a sus llamadas recursivas y nuevamente Python sigue siendo el mas lento en comparación de los demás.

3.6. Quick Sort

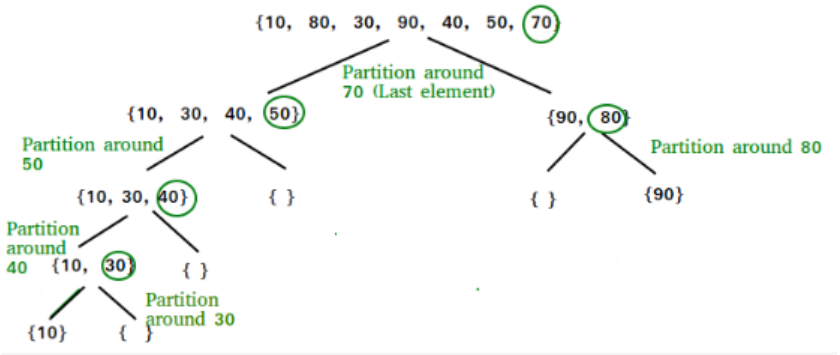
- Al igual que Merge Sort , QuickSort es un algoritmo Divide and Conquer. Elige un elemento como pivote y divide la matriz dada alrededor del pivote seleccionado.
- Hay muchas versiones diferentes de quickSort que seleccionan pivote de diferentes maneras.
  - Elija siempre el primer elemento como pivote.
  - Elija siempre el último elemento como pivote (implementado a continuación)
  - Elija un elemento aleatorio como pivote.
  - Elija la mediana como pivote.
- El proceso clave en quickSort es la partición (). El objetivo de las particiones es, dada una matriz y un elemento x de la matriz como pivote, colocar x en su posición correcta en una matriz ordenada y colocar todos los elementos más pequeños (más pequeños que x) antes de x, y poner todos los elementos mayores (más grandes que x) después X. Todo esto debe hacerse en tiempo lineal.
- **Complejidad**
  - **Mejor Caso:** O(n log n)
  - **Caso Promedio:** O(n log n)
  - **Peor Caso:** O(n)
- **Estable:** No
- **Algoritmo**

```
/ * bajo -> Índice inicial, alto -> Índice final * /
quickSort (arr [], bajo, alto)
{
    si (bajo <alto)
    {
        / * pi es un índice de partición, arr [pi] ahora
           en el lugar correcto * /
        pi = partición (arr, bajo, alto);

        quickSort (arr, bajo, pi - 1); // Antes de pi
        quickSort (arr, pi + 1, alto); // Después de pi
    }
}
```

Pseudocódigo para la partición ()

```
/ * Esta función toma el último elemento como pivote, lugares
   el elemento pivote en su posición correcta en orden
    matriz y coloca todos los más pequeños (más pequeños que el pivote)
   a la izquierda del pivote y todos los elementos mayores a la derecha
   de pivote * /
partición (arr [], bajo, alto)
{
    // pivote (Elemento que se colocará en la posición correcta)
    pivote = arr [alto];

    i = (bajo - 1) // Índice de elemento más pequeño

    para (j = bajo; j <= alto- 1; j ++)
    {
        // Si el elemento actual es más pequeño que el pivote
        if (arr [j] <pivote)
        {
            i ++; // índice de incremento del elemento más pequeño
            intercambiar arr [i] y arr [j]
        }
    }
    intercambiar arr [i + 1] y arr [alto])
    volver (i + 1)
}
```

- **Ejemplo:**



- **Implementación en Python**

```
def partition(arr,low,high):
    i = ( low-1 )
    pivot = arr[high]

    for j in range(low , high):

        if   arr[j] < pivot:
            i = i+1
            arr[i],arr[j] = arr[j],arr[i]

    arr[i+1],arr[high] = arr[high],arr[i+1]
    return ( i+1 )

def quickSort(arr,low,high):
    if low < high:
        pi = partition(arr,low,high)
        quickSort(arr, low, pi-1)
        quickSort(arr, pi+1, high)
```

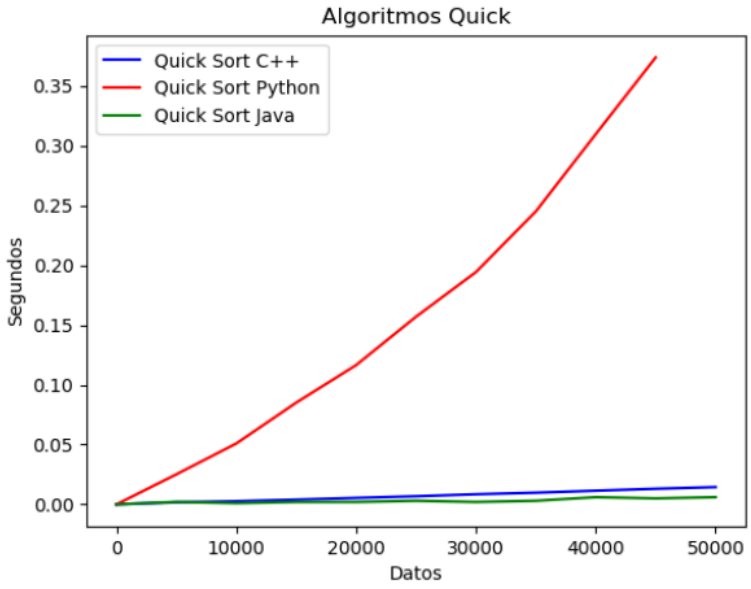
- **Implementación en C++**

```
int partition (vector<int>&arr, int low, int high)
{
    int pivot = arr[high]; // pivot
    int i = (low - 1);

    for (int j = low; j <= high - 1; j++)
    {
        if (arr[j] < pivot)
        {
            i++;
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}

void quickSort(vector<int>&arr, int low, int high)
{
    if (low < high)
    {
        int pi = partition(arr, low, high);

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

- **Implementación en Java**

```java
class QuickSort
{
    int partition(int arr[], int low, int high)
    {
        int pivot = arr[high];
        int i = (low-1);
        for (int j=low; j<high; j++)
        {
            if (arr[j] < pivot)
            {
                i++;
                int temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
            }
        }

        int temp = arr[i+1];
        arr[i+1] = arr[high];
        arr[high] = temp;

        return i+1;
    }

    void sort(int arr[], int low, int high)
    {
        if (low < high)
        {
            int pi = partition(arr, low, high);
            sort(arr, low, pi-1);
            sort(arr, pi+1, high);
        }
    }
```

- **Rendimiento del Algoritmo en los 3 lenguajes**



Este algoritmo depende mucho de su pivote pero aun hace muchas comparaciones y hace uso de mucha memoria pero es eficiente al momento de ordenar y aún Python es muy lento. La implementación predeterminada no es estable. Sin embargo, cualquier algoritmo de ordenación puede estabilizarse considerando los índices como parámetro de comparación.

3.7. Selection Sort

- El algoritmo de ordenamiento por selección clasifica una matriz al encontrar repetidamente el elemento mínimo (considerando el orden ascendente) de la parte no ordenada y colocarlo al principio.
- El algoritmo mantiene dos submatrices en una matriz determinada.
  - La submatriz que ya está ordenada.
  - Submatriz restante que no está ordenada.
- En cada iteración del orden de selección, el elemento mínimo (considerando el orden ascendente) del subconjunto sin clasificar se selecciona y se mueve al subconjunto ordenado.
- **Complejidad:** O(n)
- **Algoritmo:**

```
procedure selection sort
   list  : array of items
   n     : size of list

   for i = 1 to n - 1
   /* set current element as minimum*/
      min = i

      /* check the element to be minimum */

      for j = i+1 to n
         if list[j] < list[min] then
            min = j;
         end if
      end for

      /* swap the minimum element with the current element*/
      if indexMin != i  then
         swap list[min] and list[i]
      end if
   end for

end procedure
```

- **Ejemplo:**

```
arr [] = 64 25 12 22 11

// Encuentra el elemento mínimo en arr [0 ... 4]
// y colocarlo al principio
11 25 12 22 64

// Encuentra el elemento mínimo en arr [1 ... 4]
// y colocarlo al comienzo de arr [1 ... 4]
11 12 25 22 64

// Encuentra el elemento mínimo en arr [2 ... 4]
// y colocarlo al comienzo de arr [2 ... 4]
11 12 22 25 64

// Encuentra el elemento mínimo en arr [3 ... 4]
// y colocarlo al comienzo de arr [3 ... 4]
11 12 22 25 64
```

- **Implementación en Python**

```python
def selectionSort(arr):
    for i in range(len(arr)):
        min_idx = i
        for j in range(i+1, len(arr)):
            if arr[min_idx] > arr[j]:
                min_idx = j
        arr[i], arr[min_idx] = arr[min_idx], arr[i]
```

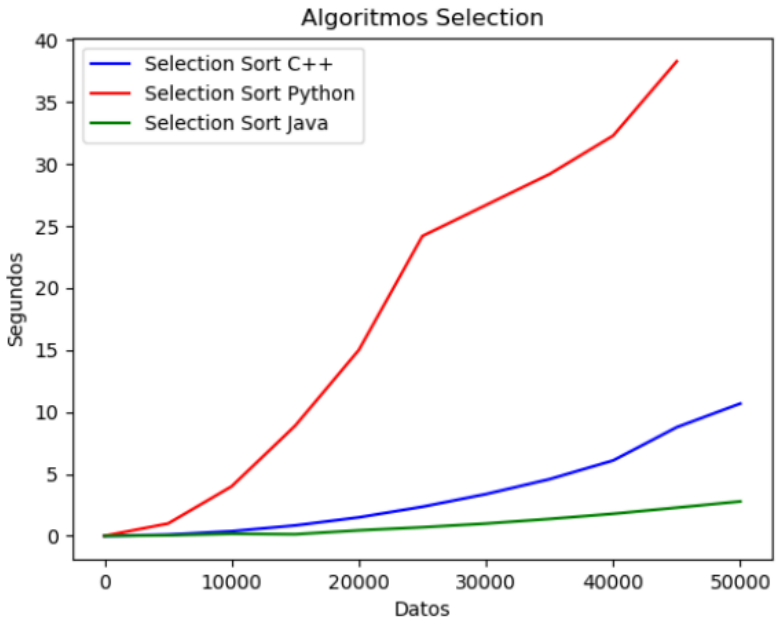
- **Implementación en C++**

```
void selectionSort(vector<int>&arr)
{
    int min_idx;
    for (int i = 0; i <arr.size()-1; i++)
    {
        min_idx = i;
        for (int j = i+1; j < arr.size(); j++)
        if (arr[j] < arr[min_idx])
            min_idx = j;
        swap(&arr[min_idx], &arr[i]);
    }
}
```

- **Implementación en Java**

```
class SelectionSort
{
    void sort(int arr[])
    {
        int n = arr.length;
        for (int i = 0; i < n-1; i++)
        {
            int min_idx = i;
            for (int j = i+1; j < n; j++)
                if (arr[j] < arr[min_idx])
                    min_idx = j;
            int temp = arr[min_idx];
            arr[min_idx] = arr[i];
            arr[i] = temp;
        }
    }
```

- **Rendimiento del Algoritmo en los 3 lenguajes**

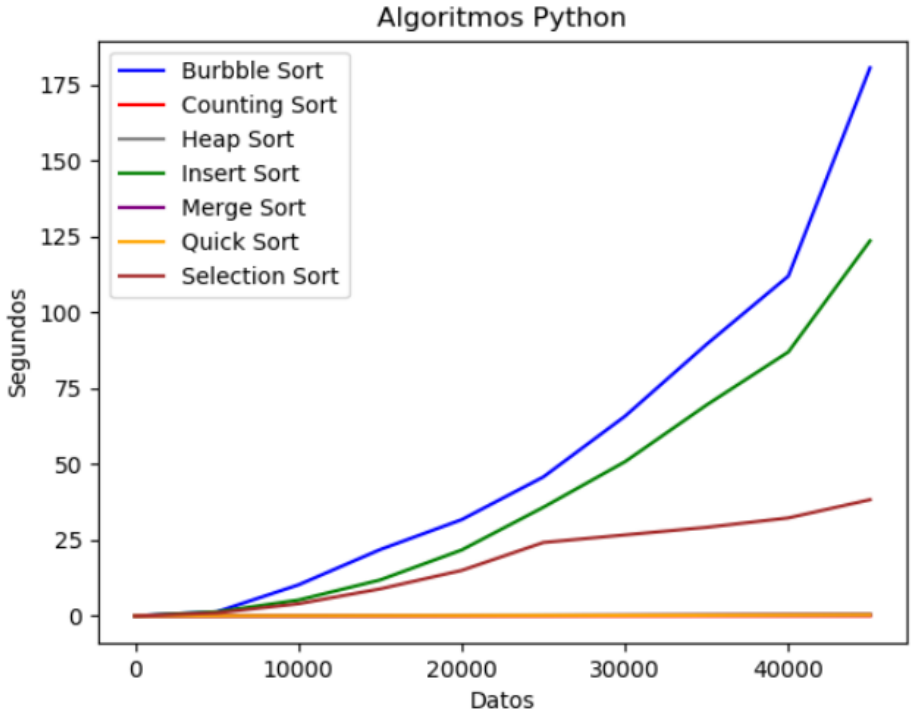


En la gráfico se puede ver que Selection Sort se demora con respecto a los otros algoritmos, sin embargo en los lenguajes Java y C++ siguen siendo mas eficientes con respecto a Python, pero aqui se observa como C++ sube en la gráfica y Java mantiene su eficiencia a pesar de la cantidad de datos.

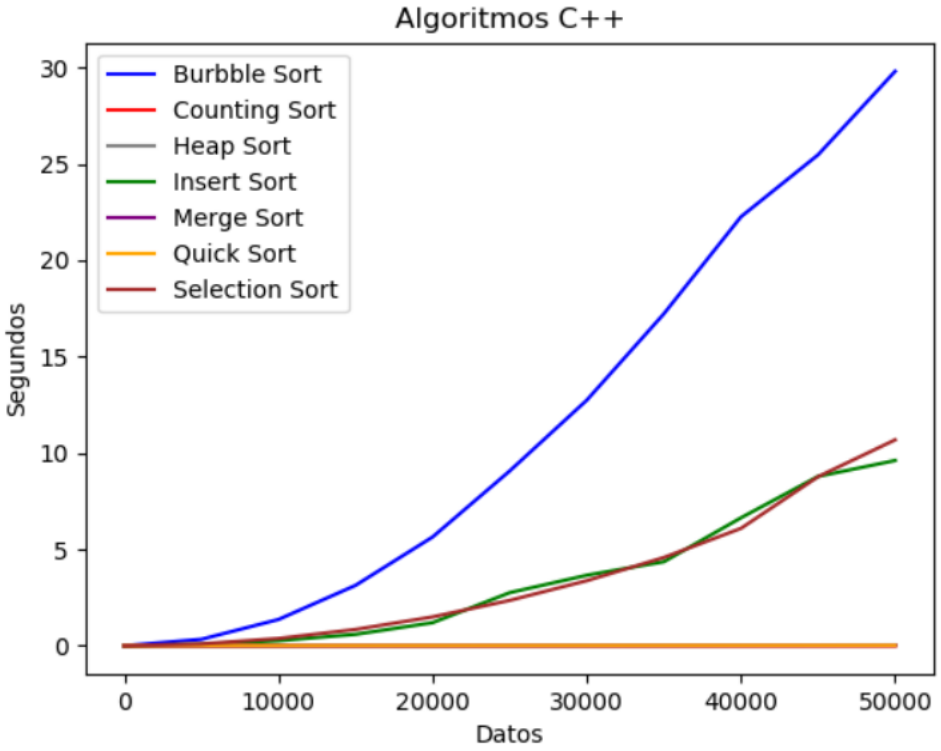
4. Conclusiones

4.1. Algortimos en Python

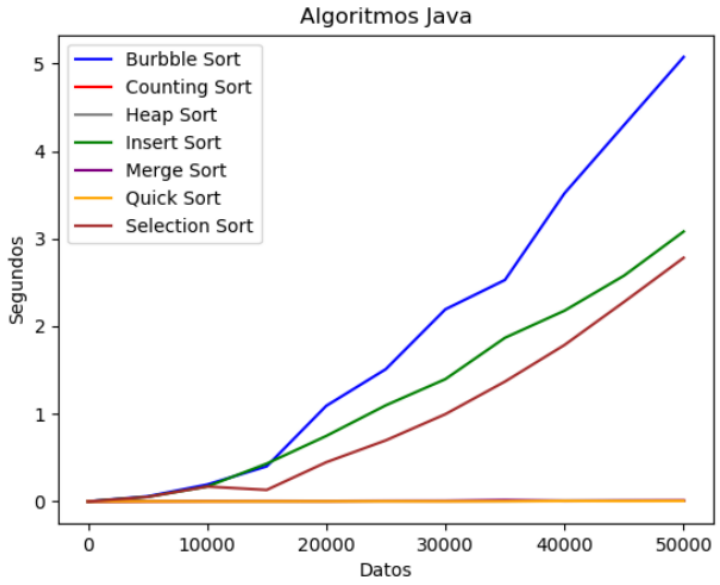
- En esta sección observaremos el Rendimiento de los Algoritmos en un lenguaje.
- **Rendimiento en Python**



- **Rendimiento en C++**



- **Rendimiento en Java**



- Se puede ver que en los 3 lenguajes usados para ete proyecto el Algoritmo con mayor Coste Computacional es el Bubble Sort, que es seguido por el Selection e Insertion Sort; Luego le siguen los Algoritmos de Divide y Venceras: Merge Sort y Quick Sort.
  Finalmente concluimos que el Algoritmo de menor Coste Computacional es uno del Tipo No Comparativo : Counting Sort para el caso de C++ y en Java sería en Quick Sort.

- **Algoritmos según su Tipo**

Nombre	**Complejidad**	**Método**
Bubblesort	O(n^2)	Intercambio
Insertion sort	O(n^2).$^{\text{en}}$ el peor de los casos"	Inserción
Counting sort	O(n+k)	No comparativo
Merge sort	O(n log n)	Mezcla
Selection sort	O(n^2)	Selección
Quick sort	Promedio: O($nlogn$), peor caso: O(n^2)	Partición
Heap sort	O(n log n)	Selección

5. Referencias

- http://ocw.uc3m.es/ingenieria-informatica/teoria-de-automatas-y-lenguajes-formales/material-de-clase-1/tema-8-complejidad-computacional
- http://ordenamientoheapsort.blogspot.com/2015/09/metodos-de-ordenamiento-los-metodos-de.html
- GeeksforGeeks
- Wikipedia
- TutorialPoint