Universidad La Salle

Facultad de Ingeniería

Escuela Profesional de Ingeniera de Software

TRABAJO DE INVESTIGACIóN

Comparación de Tiempo de Ejecución de Python, C++ y Goland

DOCENTE : Vivente Enrique Machaca Arceda

CURSO : Fundamento de Lenguaje de Programación

ALUMNOS:
Vilca Tapia Gary Jamil
Linares Salinas Kevin Joel

Arequipa - Perú
2022

Comparación de los Tiempo de Ejecución de Python, C++ y Goland

Fecha: 16 de junio del 2022

1. Abstract:

En este trabajo se muestra cómo funciona los algoritmos Quick Sort y Selection Sort, tanto en tres lenguajes diferentes. El objetivo de estas pruebas es ver cómo los lenguajes de programación pueden ejecutarse y mostrar cómo los algoritmos de ordenamiento en la fase de ejecución pueden arrojar resultados satisfactorios al momento de utilizarlos. Para poder ejecutar los algoritmos de ordenamiento se usaron tres lenguajes de programación diferentes (Python, C++ y Goland), un único computador para la ejecución y un entorno de procesamiento apto para estos tres lenguajes, la mayoría de la implementación se hizo en el sistema operativo Windows. Los resultados obtenidos permiten concluir que a través de diferentes lenguajes de programación se logra disminuir el tiempo de procesamiento de un programa de esta naturaleza.

2. Introducción:

Actualmente, el tiempo de ejecución de un algoritmo se considera como una base necesaria para optar por dicha solución para un problema en específico. Dicha ejecución de cualquier proyecto de desarrollo de software que se considere serio, y que necesite sustentarse con algo más que la correcta ejecución del algoritmo, sino que también es importante el tiempo de ejecución del mismo. Esto se debe a que un algoritmo es más eficiente si y sólo si él está optimizado en todos los aspectos y en consecuencia el tiempo de ejecución es el menor posible.

Para alcanzar un menor tiempo, se pueden hacer uso de varias técnicas y estilos de programación, incluso el lenguaje de programación en el que estemos trabajando tiene mucha mayor importancia de lo que pensábamos, esto se debe a que algunos pueden ser compilados, interpretes o mixtos.

En este trabajo de investigación, pondremos a prueba dos algoritmos de ordenamiento muy conocidas, QuickSort y SelectionSort, cada algoritmo implementado en 3 lenguajes con propiedades diferentes, estos son: Python, C++ y Goland. Analizaremos los tiempos de ejecución arrojados y los graficaremos con el fin de comprobar la eficiencia de los diferentes entornos de programación.

3. Algoritmos:

Como se mencionó anteriormente, los algoritmos que trabajaremos son Selection Sort y Quick Sort. A continuación explicaremos cada uno de los algoritmos:

3.1. Selection Sort:

El algoritmo divide la lista de entrada en dos partes: una sublista ordenada de elementos que se construye de izquierda a derecha en el frente (izquierda) de la lista y una sublista de los elementos restantes sin ordenar que ocupan el resto de la lista. Inicialmente, la sublista ordenada está vacía y la sublista no ordenada es la lista de entrada completa. El algoritmo procede encontrando el elemento más pequeño (o el más grande, según el orden de clasificación) en la sublista sin clasificar, intercambiándolo con el elemento sin clasificar más a la izquierda (poniéndolo en orden) y moviendo los límites de la sublista un elemento a la derecha.

La eficiencia temporal de la ordenación por selección es cuadrática, por lo que hay una serie de técnicas de ordenación que tienen una mayor complejidad temporal que la ordenación por selección. Una cosa que distingue a la ordenación por selección de otros algoritmos de ordenación es que realiza el mínimo número posible de intercambios, $O(n-1)$ en el peor de los casos y en el caso promedio la complejidad toma un valor de $O(n^2)$.

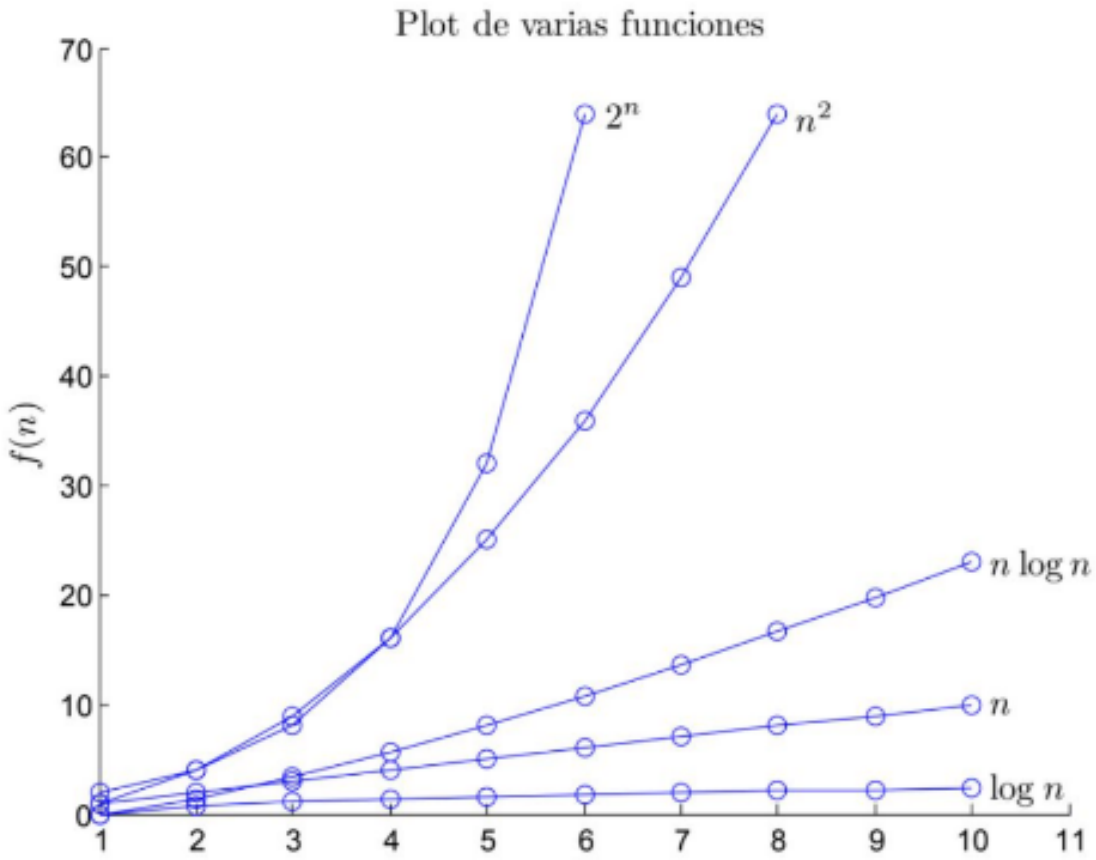
La ordenación por selección se destaca por su simplicidad y tiene ventajas de rendimiento sobre algoritmos más complicados en ciertas situaciones, particularmente donde la memoria auxiliar es limitada.

3.2. Quick Sort:

Quicksort es un algoritmo de divide y vencerás . Funciona seleccionando un elemento 'pivote' de la matriz y dividiendo los otros elementos en dos sub-matrices, según sean menores o mayores que el pivote. Por esta razón, a veces se le llama clasificación de intercambio de partición. A continuación, los subconjuntos se ordenan recursivamente . Esto se puede hacer en el lugar , lo que requiere pequeñas cantidades adicionales de memoria para realizar la clasificación.

Quicksort es una ordenación por comparación , lo que significa que puede ordenar elementos de cualquier tipo para los que se define una relación "menor que"(formalmente, una orden total). Las implementaciones eficientes de Quicksort no son una ordenación estable , lo que significa que no se conserva el orden relativo de los elementos de ordenación equivalente.

El análisis matemático de Quicksort muestra que, en promedio , el algoritmo toma $O(n \log n)$ comparaciones para ordenar n elementos. En el peor de los casos , hace $O(n^2)$ comparaciones.



Método de ordenación	Complejidad
Burbuja	O(n^2)
Selección	O(n^2)
Inserción	O(n^2)
Shell	O($n^{1.2}$)
Mezcla	O(n log$n$)
Quicksort	O(n log$n$)

4. Implementación:

4.1. Python:

- Selection Sort:

```python
from random import randint
from timeit import default_timer
import time


def SelectionSort (array):
    for i in range(len(array)):
        min_idx = i
        for j in range(i+1, len(array)):
            if array[min_idx] > array[j]:
                min_idx = j
        array[i], array[min_idx] = array[min_idx], array[i]
    for i in range(len(array)):
        #print(array[i])
        print("%d" %array[i],end=" ")

def llenar_lista(n):
    lista=[]
    for i in range (n):
        lista.append(randint(1,n))
    return lista



#Crear lista de numeros
NUM = 1000
lista = llenar_lista(NUM)

inicio = time.time()
SelectionSort(lista)
fin = time.time()

resultadoTiempoS = fin - inicio
resultadoTiempoMS = resultadoTiempoS*1000000
print(" ")
print("PRUEBA 1: LISTA DE TAMA O:", NUM)
print("TIEMPO EN MICROSEGUNDOS => ",resultadoTiempoMS)
```

Listing 1: python code

- Quick Sort:

```python
import time
def partition(arr,start,end):
  pivot = arr[end]
  i = start - 1
  for j in range(start, end):
    if arr[j] <= pivot:
      i = i + 1
      (arr[i], arr[j]) = (arr[j], arr[i])
  (arr[i + 1], arr[end]) = (arr[end], arr[i + 1])
  return i + 1

def quickSort(arr,start,end):
    if (start >= end):
        return
    p=partition(arr,start,end)
    quickSort(arr,start,p-1)
    quickSort(arr,p+1,end)


arr = [ 9, 3, 4, 2, 1, 8 ]
n = len(arr)
inicio = time.time()
quickSort(arr, 0, n - 1)
fin = time.time()
print(arr)
print("Execution Time:",fin-inicio)
```

Listing 2: python code

4.2. C++:

- Selection Sort:

```cpp
#include <bits/stdc++.h>
#include <ctime>
using namespace std;


void llenar_lista(int lista[], int n){
  int mayor;
  srand(time(NULL));
  int listado = n+1;
  for(int i=0; i<n; i++){
    lista[i]=rand()%listado;
  }
}

//Funcion de cambio
void swap(int *xp, int *yp)
{
    int temp = *xp;
    *xp = *yp;
```

```
    *yp = temp;
}

void selectionSort(int arr[], int n)
{
    int i, j, min_idx;

    for (i = 0; i < n-1; i++)
    {
        min_idx = i;
        for (j = i+1; j < n; j++)
        if (arr[j] < arr[min_idx])
            min_idx = j;
        swap(&arr[min_idx], &arr[i]);
    }
}

void printArray(int arr[], int size)
{
    int i;
    for (i=0; i < size; i++)
        cout << arr[i] << " ";
    cout << endl;
}

int main()
{
  int cantidad = 10000;
  int lista[cantidad];
  llenar_lista(lista, cantidad);
  //printArray(lista,cantidad);
  unsigned inicio = clock();
    selectionSort(lista, cantidad);
    unsigned fin = clock();
    double time = (double(inicio-fin)/CLOCKS_PER_SEC);
    cout << "Lista Ordenada: \n";
    //printArray(lista, cantidad);
    float FinMS = ((double)fin);
    cout<< "Tiempo de ejecucion:"<<time<<endl;
    return 0;
}
```

Listing 3: c++ code

- Quick Sort:

```
#include <iostream>
using namespace std;
#include <ctime>
unsigned t0, t1;
int partition(int arr[], int start, int end){
  int pivot = arr[end];
  int i = (start - 1);
  for (int j = start; j < end; j++) {
    if (arr[j] <= pivot) {
```

```
      i++;
      swap(arr[i], arr[j]);
    }
  }
  swap(arr[i + 1], arr[end]);
  return (i + 1);
}
void quickSort(int arr[], int start, int end){
  if (start >= end)
    return;
  int p = partition(arr, start, end);
  quickSort(arr, start, p - 1);
  quickSort(arr, p + 1, end);
}
int main(){
  int arr[] = { 9, 3, 4, 2, 1, 8 };
  int n = 6;
  t0=clock();
  quickSort(arr, 0, n - 1);
  t1 = clock();
  for (int i = 0; i < n; i++) {
    cout << arr[i] << " ";
  }
  double time = (double(t1-t0)/CLOCKS_PER_SEC);
  cout << "Execution Time: " << time << endl;
  return 0;
}
```

Listing 4: c++ code

4.3. Goland:

- Selection Sort:

```
package main

import (
  "fmt"
  "math/rand"
  "time"
)

// Generates a slice of size, size filled with random numbers
func llenar_lista(size int) []int {

  slice := make([]int, size, size)
  for i := 0; i < size; i++ {
    slice[i] = rand.Intn(size)
  }
  return slice
}

func selectionsort(items []int) {
  var n = len(items)
```

```
  for i := 0; i < n; i++ {
    var minIdx = i
    for j := i; j < n; j++ {
      if items[j] < items[minIdx] {
        minIdx = j
      }
    }
    items[i], items[minIdx] = items[minIdx], items[i]
  }
}
func main() {
  slice := llenar_lista(1000)
  //fmt.Println(slice)
  inicio := time.Now()
  //fmt.Println("\n--- Desordenada --- \n\n", slice)
  selectionsort(slice)
  //fmt.Println("\n--- Ordenada --- \n\n", slice)
  duracion := time.Since(inicio)
  fmt.Println("Tiempo en Microsegundos: ", duracion.Microseconds())
}
```

Listing 5: goland code

- Quick Sort:

```
package main

import (
  "fmt"
  "time"
)

func partition(arr []int, start, end int) int {
  var pivot int = arr[end]
  var i, j int = (start - 1), start
  for j < end {
    if arr[j] <= pivot {
      i++
      arr[i], arr[j] = arr[j], arr[i]
    }
    j++
  }
  arr[i+1], arr[end] = arr[end], arr[i+1]
  return i + 1
}
func quickSort(arr []int, start, end int) {
  if start >= end {
    return
  }
  var p int = partition(arr, start, end)
  quickSort(arr, start, p-1)
  quickSort(arr, p+1, end)
}
func main() {
  arr := []int{9, 3, 4, 2, 1, 8}
```

```
  var n int = 6
  start := time.Now()
  quickSort(arr, 0, n-1)
  timeElapsed := time.Since(start)
  for i := 0; i < n; i++ {
    p := arr[i]
    fmt.Println(p)
  }
  fmt.Printf("Execution Time: %s", timeElapsed.Microseconds())
}
```

Listing 6: goland code

Para capturar los resultados, hemos diseñado algoritmos capaces de crear un archivo de texto plano que almacene el arreglo con el que trabajaremos. A continuación, dicho código:

```
from random import randint
import numpy as np

def llenar_lista(n):
    lista=[]
    for i in range (n):
        lista.append(randint(1,n))
    return lista
NUM = 1000
lista = llenar_lista(NUM)
print (lista)
archivo= open('datos.txt','w')
archivo.write('%s'%lista)
archivo=open('datos.txt','r')
dades=archivo.read()
```

Listing 7: python code

5. Resultados:

A continuación, los resultados obtenidos luego de ejecutar cada algoritmos 5 veces en la misma máquina:

5.1. Tablas Comparativas:

Promedio	Python	C++	Go
prueba 100	8.6	4.8	3.6
prueba 1000	125.4	50.6	51.6
Prueba 2000	724.2	98.4	112.4
Prueba 3000	997.4	161.6	187.2
Prueba 4000	1054.6	214.2	253.8
Prueba 5000	1882.2	340.6	296.2
Prueba 6000	2051.8	409.4	382.4
Prueba 7000	2544.2	468.2	428.4
Prueba 8000	2947.2	523.8	504.4
Prueba 9000	2960	655.8	652
Prueba 10000	3634.2	696	728.8
Prueba 20000	7478.4	1514.8	1724.6
Prueba 30000	11170	2203.8	2254.8
Prueba 40000	16771.6	3090.4	4022
Prueba 50000	20620.4	3913.4	3536

Tabla 1: Promedio QuickSort

Promedio	Python	C++	Go
prueba 100	11.6	8	10.4
prueba 1000	641.6	386	1128.2
Prueba 2000	2327	1459.6	4464.8
Prueba 3000	5144.6	2583.2	8950.2
Prueba 4000	9272	4068.8	38805
Prueba 5000	14185.4	6086.8	28182.2
Prueba 6000	20423.4	9592.2	45195.4
Prueba 7000	27759.4	14713.2	54200
Prueba 8000	35948.6	17671.2	108515.8
Prueba 9000	46738.8	19037.2	137618.2
Prueba 10000	56727.6	24898.2	144222.8
Prueba 20000	231739	104767.6	689240.4
Prueba 30000	521820	109508	1312794
Prueba 40000	968394.6	85513.6	2204782.2
Prueba 50000	1505606.6	702579.6	3223559.8

Tabla 2: Promedio SelectionSort

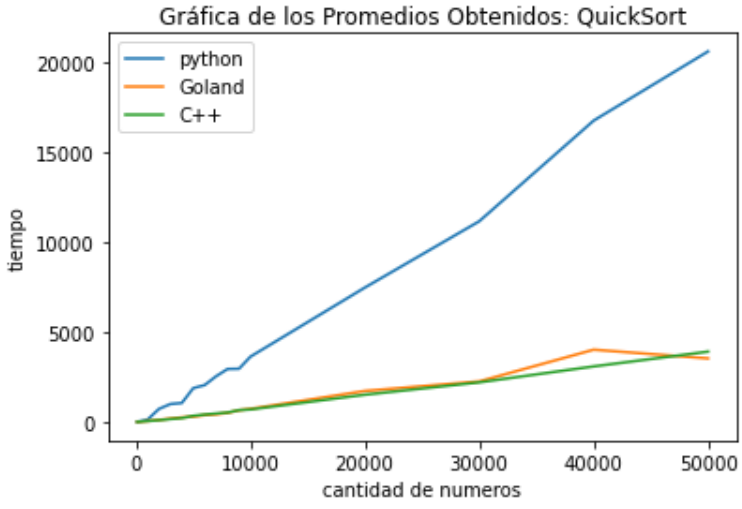
Desviación	Python	C++	Go
prueba 100	1.140175425	0.836660027	0.547722558
prueba 1000	13.01153335	12.95376393	4.979959839
Prueba 2000	138.9863303	5.594640292	13.24009063
Prueba 3000	46.80064102	7.602631123	16.88786547
Prueba 4000	22.41205033	7.949842766	30.41710045
Prueba 5000	125.0167989	11.23832728	21.56849554
Prueba 6000	57.46912214	22.68920448	45.83448483
Prueba 7000	308.7801807	29.52456604	58.12314513
Prueba 8000	456.7643813	32.01093563	76.49052229
Prueba 9000	187.2605137	66.6610831	68.42879511
Prueba 10000	375.2401897	47.09033871	162.3135854
Prueba 20000	287.1224477	95.41331144	534.9124227
Prueba 30000	763.6835077	163.660319	256.8291261
Prueba 40000	501.6540641	237.0259479	1324.914714
Prueba 50000	457.2540869	130.607427	540.2712282

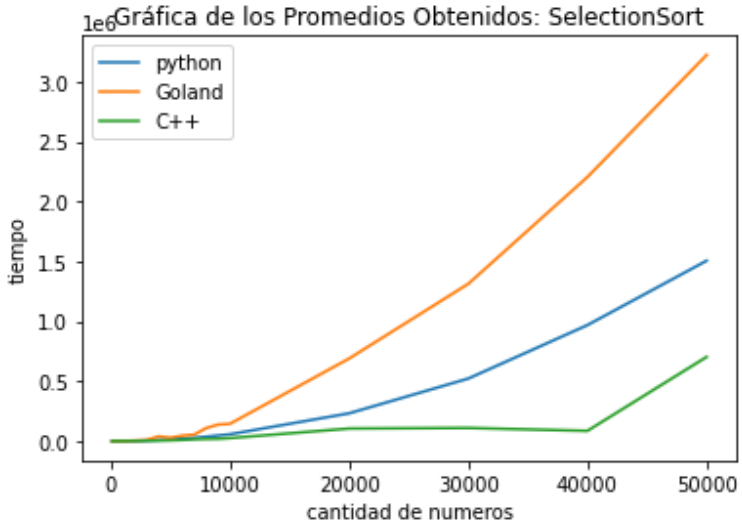
Tabla 3: Desviación QuickSort

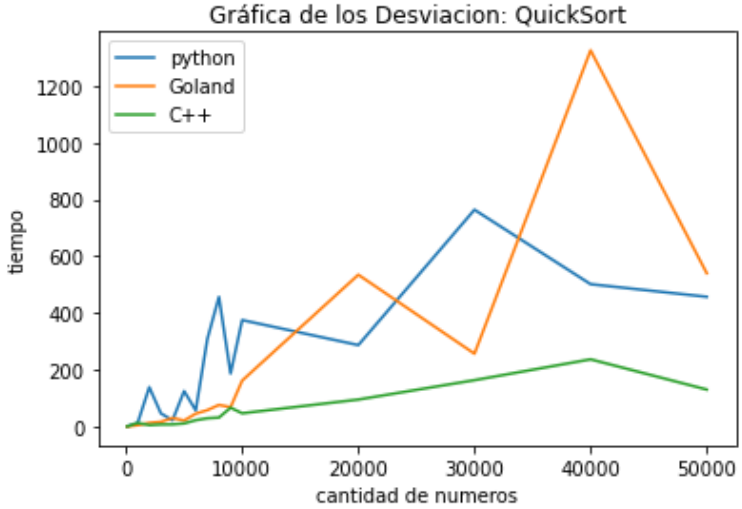
Desviacion	Python	C++	Go
prueba 100	4.159326869	0.707106781	2.50998008
prueba 1000	51.15955434	70.72835358	341.7787296
Prueba 2000	31.65438358	237.7526025	2257.861754
Prueba 3000	51.14000391	333.1961885	2983.30751
Prueba 4000	97.49102523	631.8806849	34490.72102
Prueba 5000	222.1594022	707.3996042	29988.1491
Prueba 6000	113.7949911	2099.758486	31987.69164
Prueba 7000	278.144387	2446.426925	33805.23364
Prueba 8000	149.1100265	3402.368249	32012.51531
Prueba 9000	798.6805995	2624.356816	44099.19128
Prueba 10000	498.0796121	2449.988306	37732.82274
Prueba 20000	1031.575736	27199.95901	106484.651
Prueba 30000	5725.780951	31136.32812	151179.4078
Prueba 40000	2155.831812	10903.85188	352216.367
Prueba 50000	7948.415742	83483.06285	384072.5601

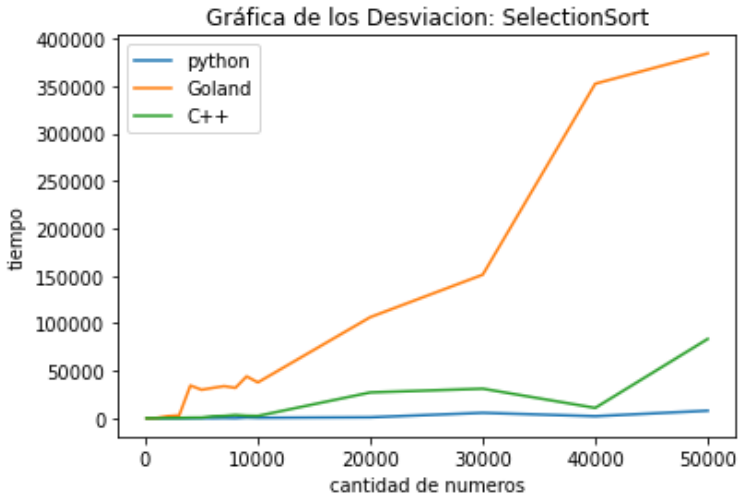
Tabla 4: Desviación SelectionSort

5.2. Gráficos Obtenidos:









6. Enlace del Repositorio GitHub:

https://github.com/kjoel2001/Practica16

7. Conclusión:

En la actualidad, toda organización que utiliza software de calidad para agilizar las tareas y así mismo maximizar el desarrollo del trabajo, para aquello los software que utilizan deben de tener un análisis profundo y por eso necesitan pasar por un periodo de pruebas para identificar funciones específicas que debe realizar y también si el producto tiene un desempeño óptimo a las funciones que se realizan.

En este caso, hemos observado los distintos tiempos de ejecución arrojado por las implementaciones de dos algoritmos de ordenamiento: Selection Sort y Quick Sort, cada uno de ellos en tres lenguajes de programación diferentes. En conclusión, el tipo de ordenamiento depende de 4 factores, estos son: Grado de ordenamiento de un arreglo, longitud del arreglo, lógica del algoritmo para ordenar y por supuesto el lenguaje de programación que estemos utilizando para la implementación. Estos dos últimos son los más importantes; ya que, todos los lenguajes de programación están diseñados para diferentes tareas.

Hemos apreciado que Python es el más lento de todos, esto se debe a que el código se interpreta en tiempo de ejecución en lugar de ser compilado a código nativo en tiempo de compilación. A diferencia de C++ y Goland, ya que estos son lenguajes compilados; es decir, convierte el código a lenguaje máquina a medida que es ejecutado. Además, la lógica que poseen dichos algoritmos son diferentes a pesar que tienen el mismo objetivo. Ya que como mencionamos anteriormente, ambos tienen una complejidad distinta, haciendo que la comparación no sea del todo equitativa para ambos algoritmos.