Analisis de complejidad computacional de los algoritmos de ordenamiento bubble, merge, quick, counting

Carlos Arturo Moreno Tabares[1]

[1]Ingeniero de sistemas, Universidad Tecnologica de Pereira, Risaralda, Colombia

Abstract: The accumulation of data, objects or other elements it creates the need to group those in one or more containers in computing one of the data structures that work such as data containers is the array, one of the operations that have the array is the ordering of the data there are several algorithms that order the content of the array where each one has different computational complexity and performance due to certain characteristics, one is the execution time which can be estimated observing the structure of the algorithm and get its behavior in O notation, there are other characteristics that are part of the computational complexity and performance as the memory and specific variables of the problem in the case of ordering are the comparisons and exchanges these other characteristics base the O notation and the execution times that are obtained from the tests.

Palabras clave: complejidad | ordenamiento | algoritmo | computación | memoria | python | rendimiento

Resumen: La acumulación de datos, objetos u otros elementos genera la necesidad de agrupar en uno o más contenedores en computación una de las estructuras de dato que son contenedores de datos son los arreglos, una de las operaciones que tienen los arreglos es el ordenamiento de los datos hay varios algoritmos que ordenan el contenido del arreglo donde cada uno tiene diferente complejidad computacional y rendimiento a razón de ciertas caracteristicas, uno es el tiempo de ejecución el cual puede estimarse observando la estructura del algoritmo y obtener su comportamiento en notación O, hay otras caracteristicas que son parte de la complejidad computacional y rendimiento como la memoria y variables especificas del problema en el caso de ordenamiento son las comparaciones e intercambios estas otras caracteristicas fundamentan la notación O y los tiempos de ejecución que se obtengan de las pruebas.

Correspondencia: ✉*carlox216@utp.edu.co*

Introducción

Las acumulaciones de datos, objetos, individuos o cualquier otro tipo de elemento para tener un registro y/o conocimiento de lo existente es común elaborar una herramienta que permita agrupar y almacenar los datos en un contenedor así como una lista, las herramientas de agrupamiento en computación son ciertas estructuras de datos que permiten agrupar elementos como arreglos (array), listas, vectores entre otros. Cada una de estas estrcturas de dato tienen su técnica de efectuar las insersiones, recorridos, eliminación y otros métodos que gestionan esas acumulaciones de datos. Los elementos que están agrupados en un contenedor cada uno de estos ocupa un espacio el cual se identifica por medio de una enumeración esta se conoce como "indice", la enumeración le da una noción de "orden" sin importar cual estructura de dato tenga su agrupación.

Las operaciones que se pueden realizar sobre el agrupamiento de datos son menos costosas computacionalmente si tienen un orden a razón de los datos así como realizar busquedas, determinar existencia, entre otros algoritmos que se benefician del orden, existen varios algoritmos que ordenan un grupo de datos como lo son bubble sort, merge sort, quick sort, counting sort y otros, cada uno de los mencionados tiene su método para ordenar el grupo de datos esto implica que cada método tiene diferente costo de ejecución sea en tiempo. memoria o otra caracteristica que indica si es mas o menos eficientes según el caso que se presente, por medio de estategias de analisis de rendimiento de software (profiling)(1), se determina que tan eficiente es cada algoritmo con respecto a los otros a razón de la complejidad comutacional.

1 Estado del arte

Las colecciones de datos en programación más especifico los arreglos cada elemento tiene una enumeración la cual se conoce como indexación siendo este un número que indica la posición del arreglo, una de las funciones más tipicas que se da en los arreglos es conocer la existencia de un elemento en el arreglo para un arreglo desordenado su complejidad es $O(n)$ ya que en el peor de los casos si el elemento a buscar no está debe de buscar en todo el arreglo, en cambio si el arreglo estuviese ordenado por algún criterio a razón de ese criterio se puede realizar la busqueda por medio de la busqueda binaria de un elemento en el arreglo con una complejidad de $log_2(n)$ (2) es decir que la busqueda binaria es mucho más efectiva ya que gracias a que está ordenado en cada iteración de la busqueda descarta la mitad de los elementos.

En la vida cotidiana el ordenar los elementos es una actividad clave ya que suele facilitar ubicar lo que se encuentra en el conjunto, en programación existen varios métodos de ordenamiento de los cuales solo seles efectuará un analisis de complejidad computacional a los métodos Bubble sort,

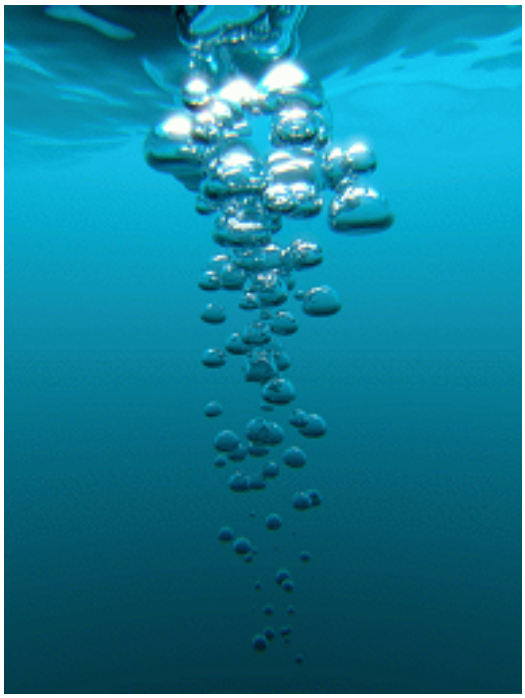

Fig. 1. Burbujas bajo el agua que relaciona al nombre del algoritmo bubble sort.

Merge sort, Quick sort y Counting sort. Las pruebas de ejecución para comprobar el rendimiento y la complejidad de los algoritmos de ordenamiento a analizar será en lenguaje python debido a que tiene una sintaxis simple de entender facilitando la implementación, sin embargo comparado con otros lenguajes puede que tenga bajo rendimiento de ejecución pero si todos son ejecutados bajo el mismo lenguaje podrá llevarse a cabo un analisis.

1.1 Bubble sort

El algoritmo de ordenamiento por burbuja (Bubble sort) su nombre dado por la revista de cientificos de la asociación por las maquinas de computo (JACM - Journal Association for Computing Machinery) basados en el algoritmo ordenamiento por intercambios (exchange sort) proveniente del libro Programming business computers autor McCracken, Daniel D. de 1959 (3), lo nombraron burbuja por que el comportmaiento del aire bajo el agua son burbujas las cuales a medida que suben a una superficie efectuan una serie de separiaciones entre las otras burbujas donde las burbujas más grandes tienden a moverse más rapido y las más pequeñas más lento generando esos intercambios con las más superioresentre todas y cada una en su camino (4) tal como se visualiza en la figura 1.
El Método para ordenar de este algoritmo es revisar desde el primer elemento hasta el n-esimo elemento de a pares el criterio de mayor o de menor, si se quiere ordenar de menor a mayor (primer elemento menor último elemento el mayor) cada que se cumpla la condición que el elemento izquierdo del par sea mayor que el derecho del par se efectua un "swap" (intercambio de los valores entre si), con dicha estrategia el elemento más mayor quedará al final de todo el arreglo eso

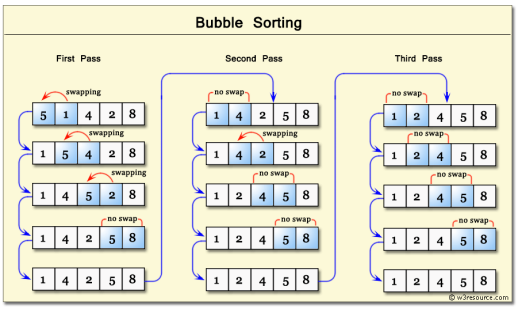

Fig. 2. Explicación visual del ordenamiento burbuja.
Fuente: w3resource

queire decir que para las siguientes iteraciones no hace falta recorrer hasta el último valor ya que se encuentra donde debe estar, la figura 2 muestra las acciones del algoritmo sobre un array en especifico.
El código fuente en lenguaje python del algoritmo ordenamiento burbuja basado en el código fuente en lenguaje c++ del artículo "Archaeological algorithmic analysis" (5) es el siguiente:

```
def bubble_sort(vector):
 n=len(vector)
 for i in range(n):
  # Check actual value with next value
  for j in range(0, n-i-1):
   # Comparision
   if(vector[j]>vector[j+1]):
    # Swapping
    vector[j], vector[j+1]=vector[j+1]\
     , vector[j]
```

De acuerdo a la estructura del algoritmo la complejidad que tiene es $O(n^2)$ ya que por cada iteración debe recorrer todo el arreglo en $n-1$ donde n es el tamaño del arreglo

1.2 Merge sort

Ordenamiento por mezcla (Merge sort) fue inventado por el matemático, físico y científico en la computación John von Neumann en 1945 (6), este algoritmo de ordenamiento da uso de la comparación con la estrategia de divide y venceras por medio de separaciones en el arreglo (slicings) hasta llegar a partes muy pequeñas y en dichas partes realizar comparaciones y realizar los respectivos cambios según el resultado de comparación hasta formar el arreglo ordenado, la figura 3 muestra las operaciones del algoritmo sobre un array en especifico.
Este algoritmo tiene una serie de variantes que pretenden reducir la complejidad del algoritmo, sin embargo el que será analizado se basa en el que se concluye del articulo de Qin song (7), el código fuente en lenguaje python correspondiente al del articulo mencionado es el siguiente:

```
# k is actual evaluate part of array
def merge(array, leftPart, rightPart, k):
 l = 0; r = 0
 left = len(leftPart)
```

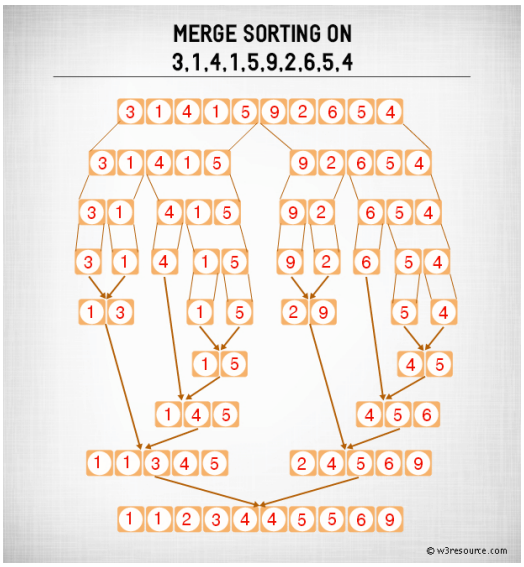

Fig. 3. Explicación visual del ordenamiento por mezcla.
Fuente: w3resource

```
    right = len(rightPart)

    while l < left and r < right:
     # Here do a comparisions
     if leftPart[l] < rightPart[r]:
      # Swap index k with l index of leftpart
      array[k] = leftPart[l]
      l += 1

     else:
      # Swap index k with r index
      # of leftpart
      array[k] = rightPart[r]
      r += 1
     k += 1

    if l < left:
     array[k:k+left-l]=leftPart[l:]
     k += left-l

    if r < right:
     array[k:k+right-r]=rightPart[r:]

def sort(array, left, right):

    if left >= right: return

    middle = int ((left + right )/ 2)

    sort(array, left, middle)
    sort(array, middle + 1, right)

    # array slice in python is [from:to]
    # where from is exactly index and to
    # is n-1 where n is to value

    # Takes in array from left to middle
```

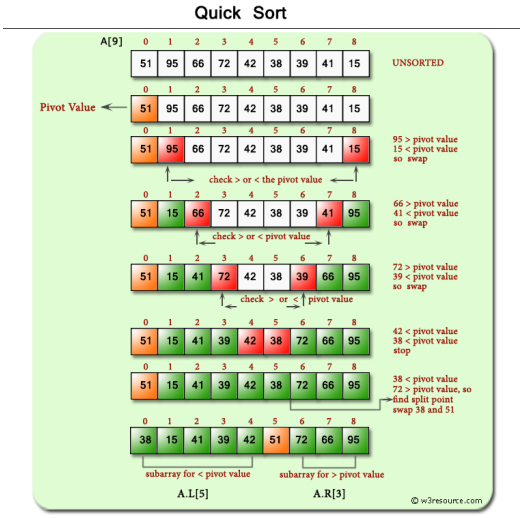

Fig. 4. Explicación visual del ordenamiento rápido.
Fuente: w3resource

```
    leftPart = array[left: middle + 1]

    # Takes in array from middle+1 to right
    rightPart = array[middle + 1: right + 1]

    merge(array, leftPart, rightPart, left)
```

De acuerdo a la estructura del código fuente anterior la complejidad promedio del algoritmo es $O(nlog_2(n))$ debido a que en el peor de los casos cada separacón del arreglo deba todos realizar un intercambio y en el algoritmo dicha separación es por mitades a todo el arreglo donde n es la cantidad de elementos del arreglo.

1.3 Quick sort

El ordenamiento rápido (Quick sort) fue presentado por el cientifico Británico Tony Hoare en 1962 (8), este algoritmo da uso de la estrategia de divide y venceras, la estrategia que utiliza el algoritmo para ordenar es utilizar un valor pivote con el cual va comparar un elemento derecho y un elemento izquierdo y dependiendo si es mayor o menor respecto al pivote (según si a como se ordene mayor o menor) serán los intercambios entre dichos elementos izquierdo y derecho, la figura 4 muestra las operaciones del algoritmo sobre un array en especifico.
Este algoritmo tiene diferentes versiones donde su variación va en como utilizar el pivote y/o su criterio de selcción, el que será analizado es donde se generan separaciones del arreglo segmentando por un vlaor de izquierda y derecha y en cda uno de estos se tiene un valor pivote el cual es elejido como el valor del medio entre los limites izquierdo y derecho de comparación, esto se efectua hasta que parte izquierda y derecha se "encuentran" (9), el código fuente en lenguaje python correspondiente a lo descrito es el siguiente.

```
def swap(array, a, b):
    array[a], array[b] = array[b], array[a]
```

```
def quickSort(array, left, right):
 # set ref for integer values

 if left >= right: return

 pivot = int((left + right) / 2)

 l = left
 r = right

 while True:
  # Determine l and r for swaping
  while array[l] < array[pivot]: l += 1
  while array[r] > array[pivot]: r -= 1

  if l == r: break

  # swap(array, l, r)
  array[l], array[r] = array[r], array[l]

  if l == pivot:
   pivot = r
   l += 1
  elif r == pivot:
   pivot = l
   r -= 1
  else:
   l += 1
   r -= 1

 quickSort(array, left, pivot-1)
 quickSort(array, pivot + 1, right)
```

Al efectuar un seguimiento en el código fuente anterior se logra determinar que la complejidad para este algoritmo es $O(nlog_2(n))$ debido a que en el mejor de los casos las segmentaciones de los sub-arreglos generados sean un mismo tamaño reduciendo de a mitades el arreglo hasta que se encuentren, en el algoritmo descrito como el pivote es el valor medio tiene mayor tendencia a formar sub-arreglos de un mismo tamaño

1.4 Counting sort

Algoritmo de ordenamiento por cuentas (Counting Sort) fue inventado por el cientifico en computación Harold Herbert Seward en 1954 (6) este algoritmo efectua un ordenamiento de una forma muy particular, a diferencia de otros métodos de ordenamiento no verifica si es mayor o menor respecto a otro valor, lo que hace es de cierto modo contar la cantidad de cada elemento del rango de un minimo hasta un maximo que exista en el array y por medio de este conteo que es efectuado en un arreglo auxiliar ubica en un nuevo arreglo según el conteo que haya efectuado los datos que tenga (10), la figura 5 muestra el arreglo de conteo correspondiente de un arreglo en particular a ordenar.

El algoritmo para contar requiere recorrer todo el arreglo una unica vez para luego recorrer el arreglo de conteo donde su

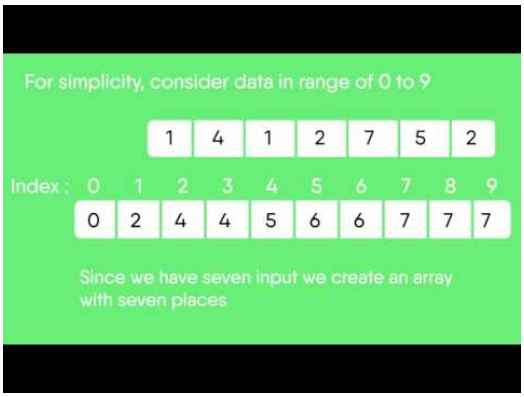

Fig. 5. Vector de conteo para ordenamiento por cuentas.
Fuente: geeksforgeeks

cantidad de elementos depende de la cantidad de valores que haya entre el valor minimo y maximo del arreglo a ordenar ya que el arreglo de conteo indica la cantidad de veces que está o no está un número dado por el indice, la versión principal del algoritmo está hecho para funcionar solo con números positivos por lo cual la siguiente implementación en lenguaje python tiene una modificación para que funcione con positivos y negativos.

```
def countingSort(array):
 _min = min(array)
 _max = max(array)
 bot = _min * -1

 if _min < 0:
  size = bot + (_max + 1)
 else:
  size = _max + 1
  bot = 0

 # get times are a value, where
 # the index is the value and
 # the value is the times are in vector
 count = [0] * size
 for item in array:
  index = bot + item
  count[index] += 1

 # get number "distance" to get a value
 # in the list from the value with index
 for i in range(1, size):
  count[i] += count[i-1]

 # set on list result from distance
 # with vector count
 result = [0] * len(array)
 for item in array:
  index = bot + item
  result[count[index] - 1] = item
  count[index] -= 1

 return result
```

En el código fuente anterior se obtiene el valor menor y mayor del vector con funciones propias de python los valores de maximo y minimo determinan el tamaño para el arreglo de conteos y para contar cuantas veces está cada uno de los valores en el arreglo debe recorrer una unica vez y luego dicha lista debe ser recorrida, por lo tanto la complejidad para este algoritmo es de $O(n+k)$ donde n es la cantidad de elementos que tiene el arreglo a ordenar y k es la cantidad de elementos que hay desde el valor minimo hasta el valor maximo del arreglo a ordenar, técnicamente la complejidad es lineal.

2 Analisis de los resultados

Los algoritmos tienen varias caracteristicas que en cierto modo permiten medir el rendimiento e indicar frente a que tipo de circunstancias es mejor un algoritmo respecto a otros, las caracteristicas que serán parte del analisis son las que miden rendimiento en general de un algoritmo y algunas que son particular en los algoritmos de ordenamiento como:

- Tiempo de ejecucicón (complejidad temporal)
- Uso de memoria (complejidad espacial)
- Cantidad de comparaciones
- Cantidad de intercambios

Para obtener un comportamiento con las caracteristicas mencionadas cada uno de los algoritmos a analizar serán sometidos a ejecuciones donde ordenaran arreglos de diferentes tamaños con su respectivo método, dependiendo del lenguaje de programación es la forma en que este genera el binario de ejecución donde a su vez cad alenguaje tiene formas distintas de medir cada una de las caracteristicas necesiarias para el analisis de rendimiento.

2.1 Software

2.1.1 Lenguaje de programación

Los algoritmos están implementados y adaptados en lenguaje **python** (los que se muestran en la sección del "Estado del arte"), python es un lenguaje de programación simple y que su funcionamiento en ejecución y memoria es dinamico debido a que utiliza una estructura de dato denominada "stack" el cual está escrito en lenguaje de programación C en donde almacena las referencias de cada una de las varaibles con respecto a los datos que apunta, el valor de los datos se almacenan en una estructura de dato "heap", dado a que tiene un comportamiento dinamico lo que hace es que por cada variable o valor nuevo determina si ya existe, en el caso que ya exista el valor no lo duplica y lo que hace es las variables apuntan a su correspondiente, a pesar de que tenga dos variables diferentes apuntando a un mismo valor los modifica de forma independiente. (11)
En las ejecuciones se da el caso de que si se tiene un arreglo de muchos elementos, cada elemento repetido ocupará un elemento más en el Stack pero no ocupará elementos adicionales en el heap, debido a este comportamiento se hace necesario que por cada ejecución que se realice en los algoritmos por cada tamaño de arreglo será el mismo arreglo inicial en cada ejecución, de esta manera se evitará distorciones en los resultados, para la toma de los datos requeridos de las diferentes caracteristicas a medir en el rendimiento (las mencionadas en la sección 2) python se debe dar uso de algunas estrategias u unos módulos que facilitan la obtención así como:

- **time:** Módulo estándar de python para acceder al reloj del sistema operativo y asi obtener el tiempo actual con el que se mide el tiempo de ejecución
- **psutil:** Módulo elaborado por Giampaolo Rodola este es capaz de administrar y obtener datos de los procesos asi como su consumo en memoria, sus procesos relacionados y obtener otro tipo de información que permita el sistema operativo obtener de cada uno de los procesos del mismo.
- **memory_profiler:** Módulo de python elaborado por Fabian Pedregosa hecho para monitorear el consumo de memoria en un proceso de python con funciones del modulo psutil, discriminando cada uno de los tipos de memoria que utiliza, puede efectuar monitoreo en una función en especifico pero solo obtiene el peak del RSS (Resident Set Size) siendo esta la memoria que utilizó de forma directa el proceso de python durante toda la ejecución.
- **tracemalloc:** Módulo estándar de python que cuenta con herramientas de depuración (debug) para medir la memoria asignada en bloques de código, esta función obtiene la memoria que tiene asignada durante la ejecución del proceso de python inclusive si está pendiente a liberar (esta memoria "reservada" se incrementa a razón de la creación de estructuras de datos de python y varia su comportamiento según la estructura).
- **Cantidad de comparaciones:** Por medio de una variable tipo entero se cuenta por cada comparación que el algoritmo efectua, para los algoritmos que implican recursion como merge sort y quick sort se utiliza una variable contenedor que pasa el valor por referencia asi como otro arreglo, para el caso particular del counting sort es por cada que cuenta
- **Cantidad de intercambios:** Por medio de una variable tipo entero se cuenta por cada intercambio que el algoritmo realiza, para los algoritmos que implican recursion como merge sort y quick sort se utiliza una variable contenedor que pasa el valor por referencia asi como otro arreglo, para el caso particular del counting sort es por cada iteración del vector counting donde forma el vector ordenado.

Los arreglos con los que se ejecutarán los algoritmos son aleatorios, cada uno de estos algoritmos su tiempo de ejecución, uso de memoria requerido para ordenar varia según

los números que tenga y el orden inicial de los mismos por lo cual para evitar alteraciones en los resultados se genera un arreglo aleatorio del tamaño a analizar y ese mismo es ejecutado cada uno de los algoritmos a analizar, el arreglo cada que se genere se colocará en un archivo el cual transitará por cada algoritmo al momento de ejecutarse, en python el paso de datos de un archivo a la ejecución del binario (file stream) y la creación de un arreglo de gran tamaño genera da uso de mucha memoria debido al comportamiento de la memoria a medida que lo va creando genera espacio en el garbage collector donde para un pc de poca RAM puede verse afectado, sin embargo una vez creado la "limpieza" del colector se ejecuta y de dicha manera todos inician una ejecución con el vector ya inicializado dando en todos las mismas condiciones de inicio.

2.1.2 Sistema operativo

Un sistema operativo es técnicamente un software que contiene y controla un conjunto de software basico para que funcione un ordenador, ente sus aplicaciones tiene unas que gestionan y controlan servicios y recursos que tenga el dispositivo en donde este implantado (12), cada sistema operativo tiene su manera de gestionar los recursos y de manejar el dispositivo en el que se encuentra esto implica que cada lenguaje de programación en la formación de sus binarios varie un poco incluyendo su comportamiento, el sistema operativo bajo el que se realizan las pruebas es en **Windows 10**, este es uno de los sistemas operativos más comunes en la sociedad y es la versión vigente de microsoft (para el 2019 aun lo es), la gestión de los procesos se basa en servicios los cuales gestiona el usuario de forma limitada y el acceso completo lo tiene el sistema pero no es accesible para el usuario. Para el lenguaje de programación python en windows 10 tiene su versión compatible con una variante en la estructura de la memoria de python, haciendo que este sea un poco más costoso en memoria que otros sistemas operativos como linux, sin embargo todos los algoritmos tendrán las mismas dificultades por lo cual se generan datos de la medida de uso de memoria en la misma escala (13), la versión del sistema operativo para el momento en que se extrajeron los datos fue Windows 10 Home Single Language - 1903 versión **18362.207** 64 bits, el criterio de selección del sistema operativo fue por compatibilidad, lo común, lo conocido y lo disponible.

2.2 Hardware

El dispositivo que se dispone para el analisis es un ROG GL553VD, este equipo es un portatil de categoria gaming diseñado para estar al tope de su rendimiento y adaptable a un sistema de refigeración (para que el rendimiento sea estable) dicho dispositivo fue fabricado por una empresa de Taiwan "ASUSTeK Computer Inc" tiene una gran variedad de productos de informatica y electronica (14), los dispositivos que ofrece de la categoria gaming la mayoria están con overclock (alcanzar mayor velocidad en la velocidad del procesador) ofreciendo un buen rendimiento de lo que el procesador puede ofrecer, las especificaciones del dispositivo para el analisis son las siguientes:

- **Procesador:** Intel(R) Core(TM) i7-7700HQ CPU @
- **Frecuencia procesador:** 2.80Hz
- **RAM:** SDRAM 8,00 GB
- **Chipset:** Intel® HM175 Express Chipset
- **Tarjeta grafica:** GTX 1500 2GB GDDR5 VRAM
- **Disco duro:** HDD 1TB 5400RPM SSH

Para el analisis de los algoritmos será ejecutado de manera secuencial, es decir no será utilizada la tarjeta grafica que dispone el dispositivo, un factor clave que afecta el rendimiento del dispositivo es la temperatura del dispositivo, si la temperatura está por encima de 80 grados celcius el rendimiento se verá reducido ante dicha situación se efectua una regulación de la temperatura y esta finalmente se sostenga y evitar cambios en el rendimiento.

2.3 Condiciones de ejecución

Para la ejecución de cada algoritmo es menestér suministrar el nombre del dataset "input python array.txt" que contiene los datos en cuestión. Estos datos están representados en una cadena de caracteres equivalente a un arreglo en python, para posteriormente ser procesado por **eval** que es una función estándar de python que creará el arreglo en memoria, el rendimiento de los algoritmos de ordenamiento dependen del tamaño del arreglo, su contenido y su orden inicial por ello las ejecuciones se llevan a cabo con un script hecho en lenguaje python que gestiona las iteraciones y crea un arreglo aleatorio el cual almacena en un archivo, el script cumple con lo siguiente:

- Arreglo de un tamaño especifico (varia tras cada iteración)
- Cada elemento del arreglo es un valor aleatorio entre 0 y 100 incluyendolos
- Al crear el arreglo de cada iteración es almacenado en un archivo
- Se ejecuta un algoritmo de ordenamiento a la vez
- Cada algoritmo de ordenamiento lee el archivo y crea el arreglo antes de contabilizar los datos
- Cada algoritmo de ordenamiento al finalizar escribe en un archivo el resultado de lo contabilizado (cada algoritmo tiene su archivo)

Dicho script de python se encarga de crear un arreglo aleatorio y de ejecutar un algoritmo por vez con dicho arreglo, una vez que ejecuta todos los algoritmos con el arreglo que generó elabora uno nuevo de un tamaño de acuerdo a la iteración siguiente, los tamaños de los arreglos aleatorios empiezan desde 20 y tienen variación de tamaño de multiplo de 10 por

vez y por cada iteración (ejecución de un mismo arreglo con todos los algoritmos a analizar), cuando alcanza el valor 100 como tamaño de arreglo su factor de cambio por iteración se incrementa en potencia de 10, es decir que ahora por cada iteración cambia en 100 por vez y cuando alcance el valor 1000 incrementa en potencia de 10 el cambio (primera vez 10^1 segunda vez 10^2, vez n 10^n), a medida que el arreglo a ordenar aumente en tamaño va a demorar más tiempo su ejecución y en tamaños muy altos puede requerir mucha memoria durante el proceso de instanciar en memoria el arreglo, por lo cual se cada algoritmo es ejecutado que algún algoritmo que uno de los algoritmos le tome más de 1800 segundos (30 minutos) en el momento que se genere una ejecución con dicha demora para las siguientes iteraciones ese algoritmo no es ejecutado, el script empieza con tamaños pequeños del arreglo y va en aumento debido a que entre más elementos tenga el arreglo mayor será el computo requerido para ordenarlo.

Otra de las condiciones de exclusión o parada que controla el script en los algoritmos que se ejecutan tras cada iteración es que en la transición de la lectura del archivo con el arreglo y la creación del arreglo en memoria de ejecución utilice más de 4GB de la memoria RAM (debido a que si el equipo agota la RAM estará muy lento), las ejecuciones terminarán en el momento que cada uno de los algoritmos a analizar superen las condiciones.

2.4 Graficas de comportamiento

Los datos para medir el rendimiento son obtenidos en un mismo dispositivo los cuales son almacenados en archivos de extensión **csv** (uno por cada algoritmo) cada que un algoritmo finaliza escribe los resultados de las ariables para medir el rendimiento (dadas en la sección 2) y agrega un salto de linea al final de su archivo csv, para garantizar que el archivo tiene los encabezados correspondientes el script al iniciar crea el archivo csv de los algoritmos con unicamente el encabezado terminando con un salto de linea (si ya existe lo elimina y vuelve a crear), cada variable para medir el rendimiento permite elaborar una grafica respecto al tamaño del arreglo permitiendo efectuar analisis de complejidad temporal (tiempo), complejidad en el espacio (memoria), y esfuerzo aplicado por comparaciones o por intercambios.

2.4.1 Complejidad temporal (Tiempo)

El tiempo es la caracteristica principal para medir el rendimiento debido a que es la espera para obtener un resultado, esta carateristica solo tiene en cuenta principalmente las ejecuciones del procesador es decir que en cada arquitectura puede variar su comportamiento de acuerdo al compilador que se utilice, el modo que gestiona la memoria y otros factores, el tiempo de ejecución depende principalmente del procesador pero el solo tiempo no logra determinar los factores externos lo afectan, la figura 6 es la grafica del tamaño arreglo contra los tiempos de ejecución de los algoritmos a analizar

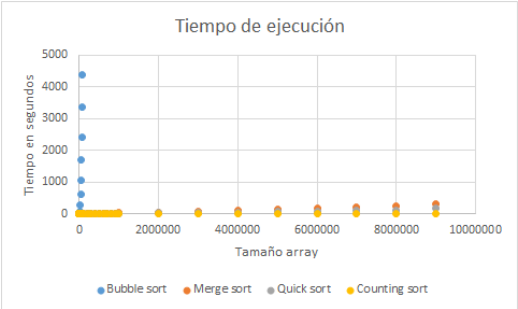

Fig. 6. Tiempos de ejecución de los algoritmos de ordenamiento.

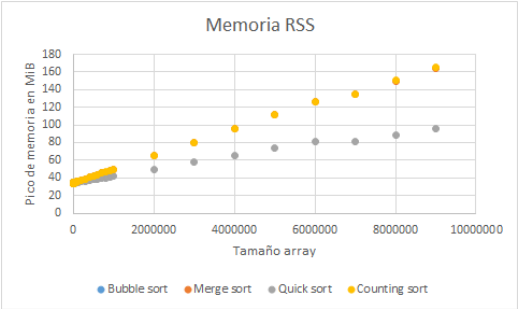

Fig. 7. Uso de memoria RSS de los algoritmos de ordenamiento.

Las gráficas de tiempo de ejecución muestran que el algoritmo Bubble sort para menos de 100000 ejecuciones le toma cerca de 4300 segundos al observar la varación de tiempo para cada tamaño del arreglo la notación O que más se adecua es $O(n^2)$ siendo cuadratico debido a que a medida que aumenta el tamaño del arreglo tarda mucho más entre iteraciones (el incremento de 10000 a 20000 es mucho menos el incremento que 20000 a 30000), los algoritmos de merge sort y quick sort tienen un comportamiento muy similar entre estos (la forma en que estos cambian por cada iteración) es una curva muy que no alcanza ser una parabola la función que más se adecua en ambos es un logaritmo al verificar la estructura del algoritmo si cumple que la complejidad en notación O es $O(n * log(n))$ siendo logaritmico, en cambio el algoritmo counting sort presenta un comportamiento lineal ya que el cambio entre iteraciones tiende a ser muy similar en cada paso con cambios pequeños debido a que counting sort crea 2 arreglos adicionales y debe recorrerlos de ahí viene a su valor k que es la cantidad diferente de elementos que tiene por lo cual la complejidad en notación O es $O(n+k)$ siendo lineal.

2.4.2 Complejidad espacial (memoria)

Medir la memoria permite determinar o dar una estimación de cuanto requiere de memoria un sistema a medida que el problema incrementa (en este caso el tamaño del arreglo) de acuerdo a la memoria disponible para el sistema cuando se elabora la ejecución cuando se agota puede afectar los tiempos de ejecución dificultando determinar en que tipo de complejdiad en el tiempo clasifica, la grafica de uso de memoria a razón del tamaño del vector son dadas en las figuras 7 y 8
En python se utilizan diferentes memorias una es **RSS** siendo esta la memoria principal que python utiliza en su espacio de ejecución, en la figura 7 se aprecia que para los algoritmos

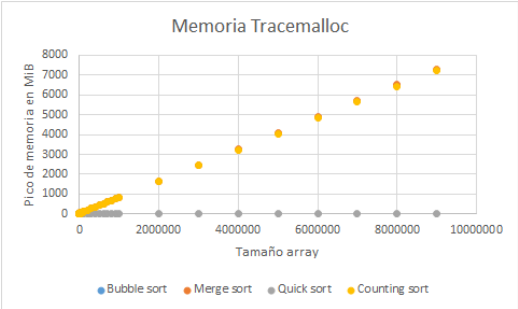

Fig. 8. Uso de memoria Tracemalloc de los algoritmos de ordenamiento.

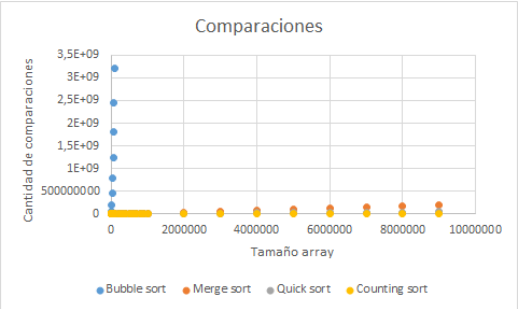

Fig. 9. Comparaciones de los algoritmos de ordenamiento.

en arreglos con una cantidad de elementos inferior a 1000000 tienden a utilizar la misma cantidad de memoria debido a que en los algoritmos recursivos como *quick sort* y *merge sort* no genera una recursion profunda (llamados pendientes en el algoritmo) a tal punto que el incremento en memoria es leve y el mismo caso es para counting sort; los algoritmos que más consumen memoria RSS son *counting sort* y *merge sort* en el caso de *counting sort* dado a que genera un segundo arreglo con la misma cantidad de elementos que tiene el arreglo a ordenar y un arreglo adicional que contiene la cantidad de elementos diferentes que tiene dicho arreglo es duplicado la cantidad de memoria requerida con un poco más, en el caso de *merge sort* por como está implementado por cada recursión del algoritmo utiliza diferentes partes del arreglo formando particiones de este como si fueran "nuevos" arreglos, el caso de *quick sort* es el que menos consume por lo que su recursión es con el mismo arreglo en todo momento y el algoritmo *Bubble sort* en lo alcanzó a ejecutar (con respecto a los demas algoritmos) se aprecia que tienden a utilizar menos memoria ya que en su ejecución secuencia no genera nuevos datos.

La memoria "completa" de la ejecución del programa en python es dado por **tracemalloc** en los resultados de la figura 8 muestran que *counting sort* utiliza una gran cantidad de memoria que aumenta de manera lineal con una pendiente elevada (gran incremento con respecto a la cantidad de elementos), se debe a que en el algoritmo cuando se está creando los arreglos counting y resultado en el lenguaje python a medida que va creando datos va reservando, como no ejecuta el garbaje collector se acumulan datos sin utilizar a medida que se va creando cada uno de los arreglos, el garbaje collector es ejecutado antes de ejecutar una nueva función, sin embargo al observar la figura 6 no resulta alterado el tiempo de ejecución (a razón de su comportamiento) debido a que usa es la memoria RSS.

2.4.3 Comparaciones

La caracteristica de comparaciones da un indicio del "esfuerzo" que aplicó el algoritmo para su labor esta caracteristica se relaciona con el tiempo de ejecución ya que es parte de las acciones que debe ejecutar el procesador, cada uno de los algoritmos de ordenamiento a analizar tienen de cierto modo algún tipo de comparación, en el caso de *bubble sort* compara un par de valores por iteración durante su recorrido, en el caso de *merge sort* sus comparaciones se dan para formar sus separaciones y luego combinar, *quick sort* realiza sus comparaciones con respecto al pivote intermedio, para el caso de *counting sort* no efectua exactamente comparaciones lo que hace es que en su arreglo de conteo suma en el indice respecto al valor que ingresa el cual puede ser considerado como una comparación, la figura 9 muestra la cantidad de comparaciones de cada algoritmo con respecto al tamaño de arreglo a ordenar.

La figura 9 evidencia el comportamiento del algoritmo *bubble sort* debido a que este evalua de forma secuencial varias veces el arreglo a ordenar donde cada vez recorre un elemento menos en todo el arreglo alcanzando más de 300000000 comparaciones en un arreglo de tamaño 9000000 explicando así el por que a dicha cantidad de elementos en el arreglo el algoritmo cuesta demasiado en tiempo para el dispositivo, en el algoritmo *merge sort* logra una cantida dde comparaciones bastante menor con relación al de *bubble sort* sin embargo si se compara con *quick sort* ha requerido 4 veces más comparaciones debido a que merge sort siempre genera divisiones del arreglo en el arreglo siguiendo el criterio de la comparación para ordenar, el algoritmo *quick sort* es mejor que *merge sort* debido a que utiliza un pivote que garantiza realizar intercambios y comparaciones partiendo de la mitad del arreglo logrando alcanzar su mejor caso la mayoria de las veces, con el algoritmo *counting sort* es quien realiza menos comparaciones debido a su estrategia será tantos elementos tiene el arreglo fundamentando por que es el mejor en tiempo de ejecución.

2.4.4 Intercambios

La caracteristica de intercambios es otra caracteristica que tambien relaciona al esfuerzo, ya que tambien es otras de las labores que lleva a cabo el procesador para que cambie los datos solo que tiene más uso en memoria por operación respecto al de las comparaciones, cada uno de los algoritmos para entregar un arreglo ordenado requiere "mover" los datos para que pueda ser ordenado, en el caso de *bubble sort* los intercambios se realizan por cada que en sus comparaciones se cumple el criterio según al ordenamiento, *merge sort* sus intercambios son efectuados en sus fragmentos del arreglo según al criterio de ordenamiento, *quick sort* se efectua un intercambio por cada que cumple su compración con respecto al pivote cumpliendo su criterio de ordenamiento y para *counting sort* sus intercambios no son realmente intercambios si no que crea un nuevo arreglo que es el arreglo que

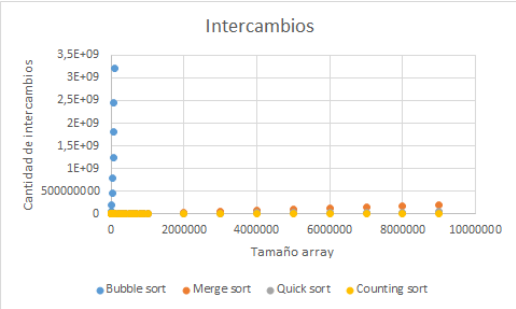

Fig. 10. Intercambios de los algoritmos de ordenamiento.

ingresa ya ordenado, por lo cual sus "intercambios" es cada inserción que efectua para crear el arreglo, en la figura 10 ilustra los intercambios que efectuó cada uno de los algoritmos de acuerdo a su estrategia.
La figura 10 confirma nuevamente el comportamiento de cada uno de los algoritmos debido a que está muy relacionado a las comparaciones, sin embargo la relación comparaciones varia según el algoritmo y el arreglo iniciar pero en el caso de *bubble sort* tiende a ser menor la cantidad de intercambios respecto a la cantidad de comparaciones, para *merge sort* tiende a ser la misma cantidad de intercambios que de comparaciones, para *quick sort* la cantidad de intercambios es cerca del doble con respecto a las comparaciones que realiza debido a que genera varias divisiones del arreglo, en el caso de counting sort es la cantidad de eleemntos que tiene el arreglo siendo igual a al cantidad de comapraciones debido a su estrategia.

3 Conclusiones

El ordenamiento de arreglos es una tarea que puede efectuarse con una variedad de métodos los cuales cada uno tiene sus diferentes ventajas y desventajas que varian de acuerdo al caso particular (el arreglo inicial) muchas de las actividades o "problemas" que se presentan tanto en la vida cotidiana como en la computación cumplen al refran *Divide y vencerás* (15) en el analisis que deja muy en claro que los algoritmos que separan el problema de ordenamiento como *merge sort* y *quick sort* son más efectivos que el *bubble sort* debido a que este no aplica ninguna separación del problema y se le crece el problema, sin embargo no siempre *separar* el problema es lo mejor posible, depende tambien de la estrategia y dicha estrategia está ligado a la estructura del problema en este caso es que contiene el arreglo y su orden inicial resultaban pequeñas variantes de tiempo de ejecución entre iteraciones que salian de su comportamiento.
Por otro lado según el problema particular puede tener soluciones aun más optimas que una estrategia del refan *Divide y vencerás* entre los algoritmos del analisis *counting sort* demuestra ser el más efectivo para los arreglos con las condiciones dadas de la sección 2.3 siendo este una estrategia "aritmetica" pero con una implicación en el consumo de la memoria, divide y vencerás es una buena estrategia para solucionarlo "rápidamente" sin requerir de un analisis exhaustivo pero varios de estos problemas pueden tener una estrategia particular de solución dado por su estructura.
Cada lenguaje de programación gestiona de manera diferente los recursos del dispositivo y este puede variar según al sistema operativo, para el caso de *Windows 10* python a medida que va creando o asignando valores si no se ejecuta despues de cierta cantidad de asignaciones una "limpieza" de la memoria que no necesita realmente (garbaje collector) puede llegar a ocupar rápidamente toda la memoria RAM del dispositivo así como se evidenció en el uso de memoria tracemalloc de *counting sort* dado en la sección 2.4.2 debido a como se crea los dos arreglos que requeire el algoritmo.

Bibliografia

1. Evelyn Duesterwald and Vasanth Bala. Software profiling for hot path prediction: Less is more. In *ACM SIGOPS Operating Systems Review*, volume 34, pages 202–211. ACM, 2000.
2. Christos H Papadimitriou. *Computational complexity*. John Wiley and Sons Ltd., 2003.
3. Daniel D McCracken. Programming business computers. 1959.
4. MEMORY COLLEGE OF ARTS AND SCIENCES - Department of Mathematics And Computer Science some sorting algorithms. `http://www.mathcs.emory.edu/~cheung/Courses/171/Syllabus/7-Sort/bubble-sort.html`. Accessed: 2019-07-07.
5. Owen Astrachan. Bubble sort: an archaeological algorithmic analysis. In *ACM SIGCSE Bulletin*, volume 35, pages 1–5. ACM, 2003.
6. Donald Knuth. Sorting and searching. *The art of computer programming*, 3:513, 1998.
7. Song Qin. merge sort algorithm. *Florida Institute of Technology*, 2014.
8. Charles AR Hoare. Quicksort. *The Computer Journal*, 5(1):10–16, 1962.
9. Joseph JaJa. A perspective on quicksort. *Computing in Science & Engineering*, 2(1):43, 2000.
10. Keshav Bajpai and Ashish Kots. Implementing and analyzing an efficient version of counting sort (e-counting sort). *International Journal of Computer Applications*, 98(9), 2014.
11. Gergö Barany. Python interpreter performance deconstructed. In *Proceedings of the Workshop on Dynamic Languages and Applications*, pages 1–9. ACM, 2014.
12. Abraham Silberschatz, Greg Gagne, and Peter B Galvin. *Operating system concepts*. Wiley, 2018.
13. Andrew S Tanenbaum and Albert S Woodhull. *Operating systems: design and implementation*, volume 68. Prentice Hall Englewood Cliffs, 1997.
14. Willy C Shih, Chintay Shih, Hung-Chang Chiu, Yi-Ching Hsieh, and Ho Howard Yu. Asustek computer inc. eee pc (a). 2008.
15. Sumit Mishra, Sriparna Saha, Samrat Mondal, and Carlos A Coello Coello. A divide-and-conquer based efficient non-dominated sorting approach. *Swarm and evolutionary computation*, 44:748–773, 2019.