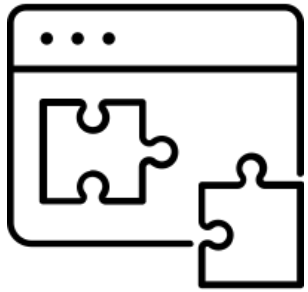
ANÁLISIS DE ALGORITMOS DE ORDENAMIENTO

PROYECTO FINAL

MANUELA ACOSTA FAJARDO Y MARÍA PAULA GAVIRIA

PLANTEAMIENTO DEL PROBLEMA

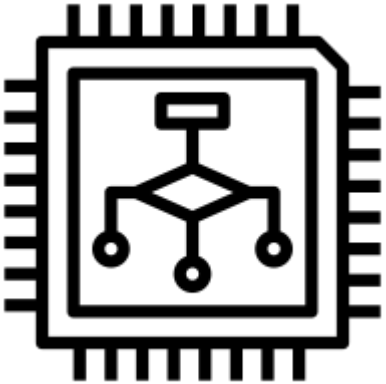
Queremos demostrar, de manera gráfica, el funcionamiento de diez diferentes algoritmos de ordenamiento, trabajando con vectores de números enteros, de forma que se evidencie cuál de todos es más efectivo y más rápido en cada situación específica.

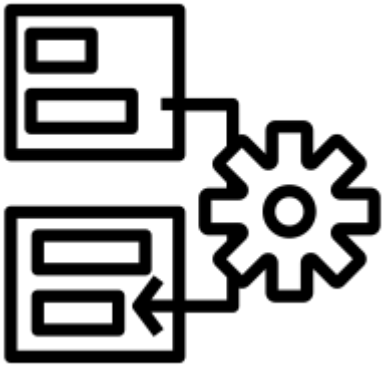


RESOLUCIÓN DEL PROBLEMA PLANTEADO

1. ELECCIÓN DE LOS ALGORITMOS

Algoritmos seleccionados para estudiar y analizar:





1. Heap Sort
2. Counting Sort
3. Selection Sort
4. Bubble Sort
5. Insertion Sort
6. Quick Sort
7. Merge Sort
8. Shell Sort
9. Gnome Sort
10. Cocktail Sort

2. CREACIÓN DE NUESTRA CLASE ‘VECTOR’

Para el análisis de los algoritmos seleccionados, decidimos crear nuestra propia estructura de datos, una clase de vectores de números enteros.

Sus métodos y atributos son los siguientes:

```
private:
        int *array;
        int capacity;
        int count;
        void expandCapacity();
```

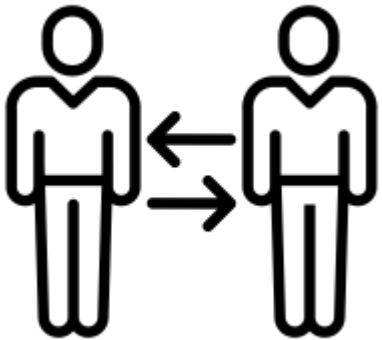
2. CREACIÓN DE NUESTRA CLASE ‘VECTOR’

```
public:
        Vec();
        ~Vec();
        int getsize();
        bool empty();
        void set(int pos, int val);
        void push_back(int x);
        void pop_back();
        int get(int pos);
        int getfront();
        int getlast();
        int getmax();
        int getmin();
        void display();
```

FUNCIONES ADICIONALES

Para poder trabajar con los algoritmos seleccionados a partir de nuestra clase creada, debimos adaptar algunas funciones adicionales, principalmente, la función ‘swap’, trabajada en clase, adaptada a nuestra clase de vectores.

```
void swap(int i, int j, Vec& vector){
 int temp = vector.get(i);
 vector.set(i, vector.get(j));
 vector.set(j, temp);
}
```



3. ADAPTACIÓN Y EXPLICACIÓN DE LOS ALGORITMOS

Una vez seleccionados los algoritmos con los que trabajaríamos, y habiendo creado nuestra propia clase de vectores, adaptamos cada uno de éstos a la clase.

Además, analizamos cómo trabajaba cada uno de los algoritmos.

HEAP SORT

Este es un algoritmo de comparación basado en una estructura de datos Binary Heap. La forma en que trabaja es encontrando el máximo elemento del vector, y ubicándolo al final del mismo. El algoritmo repite el proceso con los elementos restantes, hasta ordenar por completo el vector.

HEAP SORT

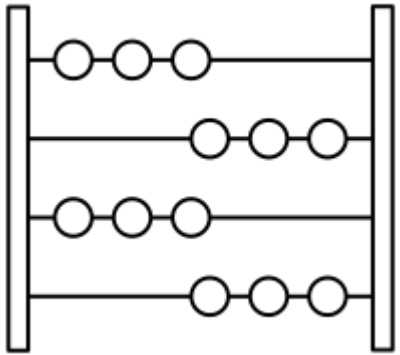
```
void heapify(Vec& vector, int n, int i){
 int largest = i;
 int l = 2*i + 1;
 int r = 2*i + 2;
 if (l < n && vector.get(l) > vector.get(largest)) largest = l;
 if (r < n && vector.get(r) > vector.get(largest)) largest = r;
 if (largest != i){
   swap (i, largest, vector);
   heapify(vector, n, largest);
 }
}
```

HEAP SORT

```
void heap_sort(Vec& vector, int n){
 for (int i = n/2-1; i>=0; i--){
   heapify(vector, n, i);
 }
 for (int i = n-1; i>=0; i--){
   swap(0, i, vector);
   heapify(vector, i, 0);
 }
}
```

COUNTING SORT

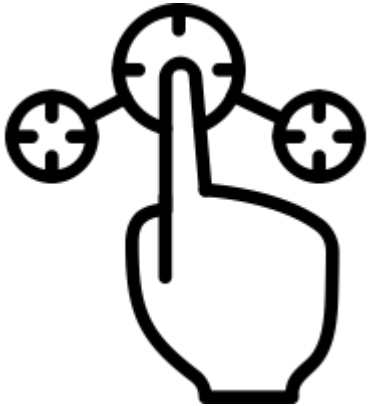
Este algoritmo ordena los elementos de un vector a partir de llaves dentro de un rango específico. Trabaja contando la cantidad de elementos que tienen distintos valores para las llaves, como una especie de Hashing. Después de esto, hace algunos cálculos aritméticos para determinar la posición de cada objeto en la secuencia de salida.



COUNTING SORT

```
void counting_sort(Vec& vector){
 map<int, int> freq;
 for(int i = 0; i < vector.getsize(); i++){
   freq[vector.get(i)]++;
 }
 int i = 0;
 for(auto it: freq){
   while(it.second--) vector.set(i++, it.first);
 }
}
```

SELECTION SORT



Este algoritmo organiza el vector encontrando el menor elemento de la parte desordenada del vector, y poniéndolo al inicio del mismo.

Se mantienen dos sub-vectores mientras se realiza el ordenamiento, la parte ordenada del mismo, y la parte desordenada.

En cada iteración, el menor elemento del sub-vector desorganizado se selecciona y se ubica en el sub-vector ordenado.

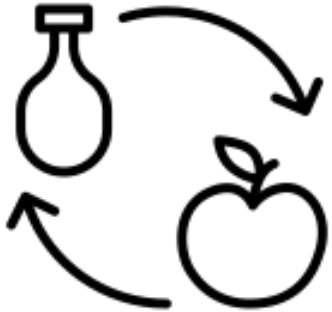
SELECTION SORT

```
void selection_sort(Vec& vector){
    for(int i = 0; i < vector.getsize()-1; ++i){
        int min_id = i;
        for(int j = i+1; j < vector.getsize(); ++j){
            if(vector.get(j) < vector.get(min_id)){
                min_id = j;
            }
        }
        swap(i, min_id, vector);
    }
}
```

BUBBLE SORT

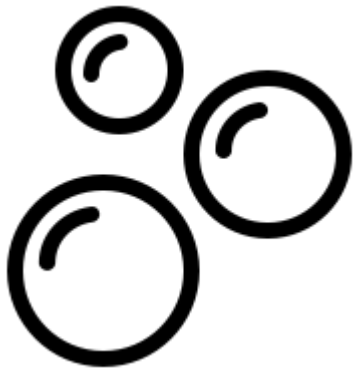
Este algoritmo es considerado como el más sencillo algoritmo de ordenamiento.

Su manera de trabajar es evaluar dos elementos adyacentes, y, en caso de no que estén en orden ascendente, los intercambia, por medio de la función ‘swap’.

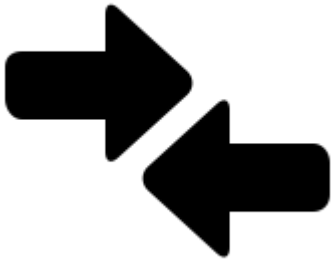


BUBBLE SORT

```
void bubble_sort(Vec& vector){
    bool swap_used = true;
    while(swap_used){
        swap_used = false;
        for(int i = 0; i < vector.getsize()-1; ++i){
            if(vector.get(i) > vector.get(i+1)){
                swap(i, i+1, vector);
                swap_used = true;
            }
        }
    }}
```



INSERTION SORT



La forma en que este algoritmo ordena los elementos de un vector es comparando un elemento, con su elemento anterior. Es decir, compara a con a-1. En caso de que a < a-1, continúa comparando a+1 con a. En caso contrario, si a > a-1, intercambia dichos elementos y compara ahora a con a-2.

INSERTION SORT

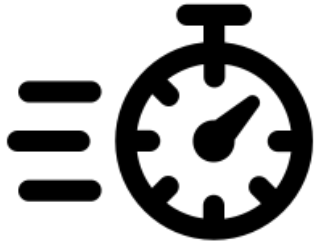
```
void insertion_sort(Vec& vector){
    for(int i = 1; i < vector.getsize(); ++i){
        int a = i;
        int b = i-1;
        while(vector.get(b) > vector.get(a)){
            swap(a, b, vector);
            a--;
            b = a-1;
        }
    } }
```

QUICK SORT

Este algoritmo es conocido como un algoritmo de ‘Divide and Conquer’. La forma en que trabaja es seleccionando un elemento del vector como pivote, y dividiendo el vector dado a partir del pivote.

La función que realiza la división es ‘partition’, y su objetivo es: dado un vector y un elemento x del vector, como pivote, y pone a x en su posición correcta en un arreglo ordenado. Además, pone todos los elementos menores que x antes que él, y los elementos mayores que x después de él.

Esto se hace en tiempo linear.



QUICK SORT

```
int partition(Vec & vector, int low, int high){
    int pivot = vector.get(high); int i = (low -1);
    for(int j = low; j <= high - 1; j++){
        if(vector.get(j) < pivot){
            i++;
            swap(i, j, vector);
        }
    }
    swap(i+1, high, vector);
    return (i+1);
}
```

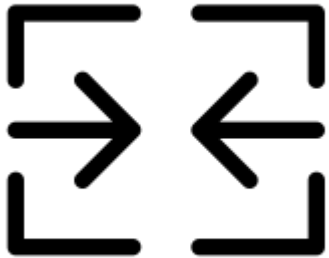
QUICK SORT

```
void quicksort(Vec & vector, int low, int high){
    if(low < high){
        int pi = partition(vector, low, high);

        quicksort(vector, low, pi - 1);
        quicksort(vector, pi + 1, high);
    }
}
```

MERGE SORT

Al igual que el algoritmo anterior, este es un algoritmo de ‘Divide and Conquer’. Éste divide el vector ingresado en dos mitades y después une las dos mitades, ya organizadas. La función ‘merge’ es utilizada para unir las dos mitades.



MERGE SORT

```
void mergesort(Vec & left, Vec & right, Vec & vector){
    int nL = left.getsize();
    int nR = right.getsize();
    int i = 0, j = 0, k = 0;
    while(j < nL && k < nR){
        if(left.get(j) < right.get(k)){
            vector.set(i, left.get(j));
            j++;
        }else{
            vector.set(i, right.get(k));
```

MERGE SORT

```
k++;
    }
    i++;
}while(j < nL){
    vector.set(i, left.get(j));
    j++;
    i++;
}while(k < nR){
    vector.set(i, right.get(k));
    k++; i++;
} }
```

MERGE SORT

```
void sort(Vec & vector){
    if(vector.getsize() <= 1) return;
    int mid = vector.getsize()/2;
    Vec left;
    Vec right;
    for(size_t j = 0; j < mid; j++){
        left.push_back(vector.get(j));
    }
```

MERGE SORT

```
    for(size_t j = 0; j < (vector.getsize() - mid); j++){
            right.push_back(vector.get(mid+j));
        }

        sort(left);
        sort(right);
        mergesort(left, right, vector);
}
```

SHELL SORT

Este algoritmo es una modificación de Insertion Sort. Sin embargo, contrariamente a este último, Shell Sort permite intercambiar elementos que están lejos entre sí, no necesariamente elementos consecutivos.

SHELL SORT

```
void shell_sort(Vec & vector){
    for(int gapSize = vector.getsize()/2; gapSize > 0; gapSize /= 2){
        for(int currentIndex = gapSize; currentIndex < vector.getsize(); currentIndex++){
            int currentIndexCopy = currentIndex;
            int item = vector.get(currentIndex);
            while(currentIndexCopy>=gapSize &&vector.get(currentIndexCopy-gapSize)>item){
                vector.set(currentIndexCopy, vector.get(currentIndexCopy - gapSize));
                currentIndexCopy -= gapSize;
            }
        vector.set(currentIndexCopy, item);
        }
    }
}
```

GNOME SORT

Este algoritmo está basado en el método en que un gnomo organiza las flores en su jardín. La forma en que trabaja es revisando el elemento que tiene al frente, con su elemento anterior. Si están en el orden correcto, sigue al siguiente elemento y compara con el anterior, y si no están en orden, los intercambia, y sigue. En caso de que no haya elemento anterior, se mueve al siguiente, y cuando no haya elemento siguiente, indica que ya terminó de ordenar los elementos.

GNOME SORT

```
void gnome_sort(Vec& vector, int sz){
   int index = 0;
 while(index < sz){
   if(index == 0) index++;
   if (vector.get(index) >= vector.get(index-1)) index++;
   else{
     swap(index, index-1, vector);
     index--;
   }
 } }
```

COCKTAIL SORT

Este algoritmo es una variación de Bubble Sort. La forma en que trabaja es, primero, viajando por el vector, de izquierda a derecha, comparando elementos adyacentes. En caso de que el elemento de la izquierda sea mayor que el de la derecha, los intercambia. Al final de las iteraciones, el mayor elemento se debe encontrar al final del vector. Después, realiza el recorrido de derecha a izquierda, revisando que los elementos adyacentes estén en su correcto orden, y, en caso de que no lo estén, cambiándolos.

COCKTAIL SORT

```
void cocktail_sort(Vec& vector, int n){
 bool swapped = true;
 int start = 0;
 int end = n-1;
 while(swapped){
   swapped = false;
   for (int i = start; i < end; i++){
     if(vector.get(i) > vector.get(i+1)){
       swap(i, i+1, vector);
       swapped = true;
     } }
```

COCKTAIL SORT

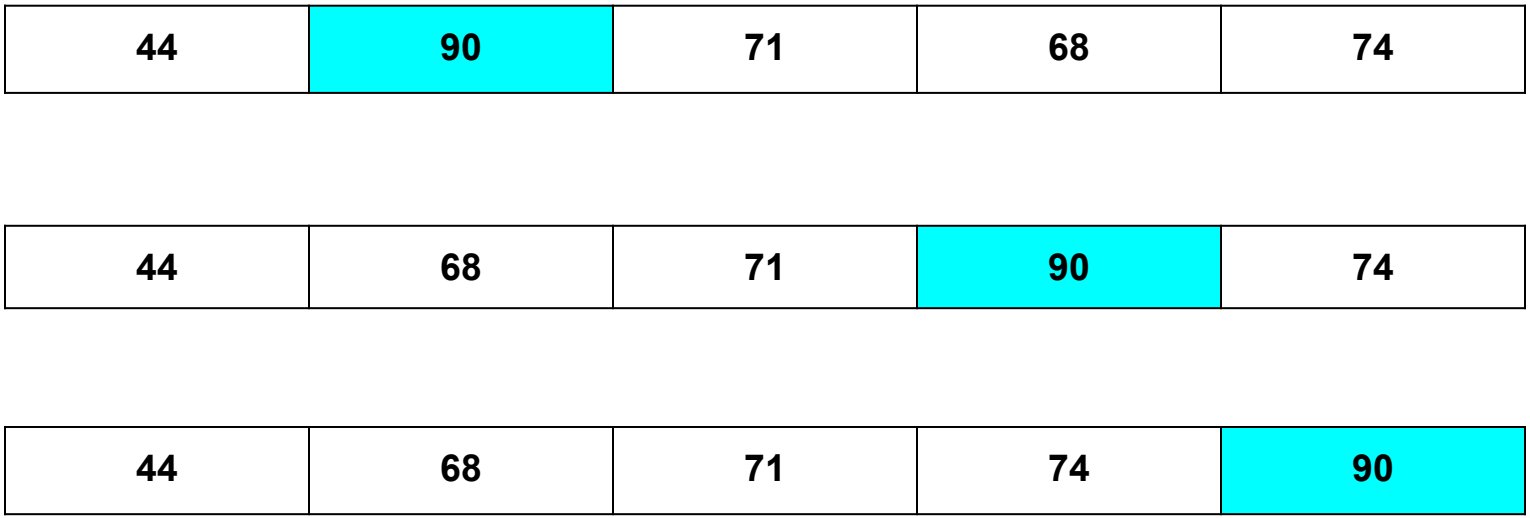
```
    if (!swapped) break;
    swapped = false;
    end--;
    for(int i = end-1; i >= start; i--){
      if(vector.get(i) > vector.get(i+1)){
        swap(i, i+1, vector);
        swapped = true;
      }
    }
    start++;
} }
```

4. VISUALIZACIÓN DE LOS ALGORITMOS

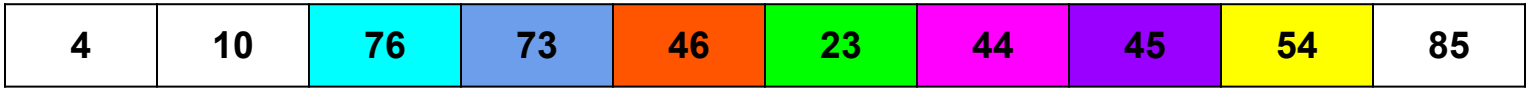
Para poder visualizar el trabajo de cada uno de los algoritmos, con los vectores creados, hicimos uso de nuestro método ‘display’ dentro del código de éstos, de forma que en cada cambio que se hiciera, se imprimiera el vector, y pudiéramos entender la forma en que el algoritmo los ordenaba.

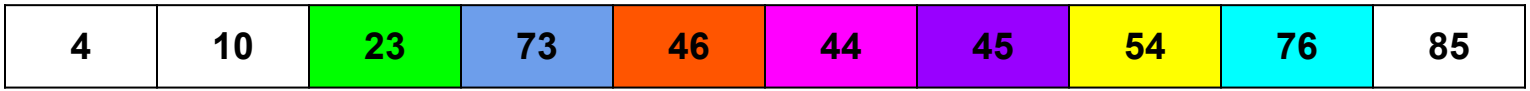
A continuación mostraremos algunos ejemplos...

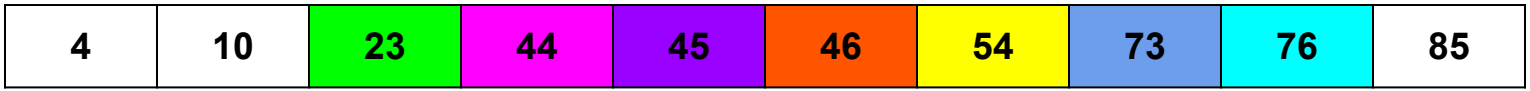
VISUALIZACIÓN SELECTION SORT



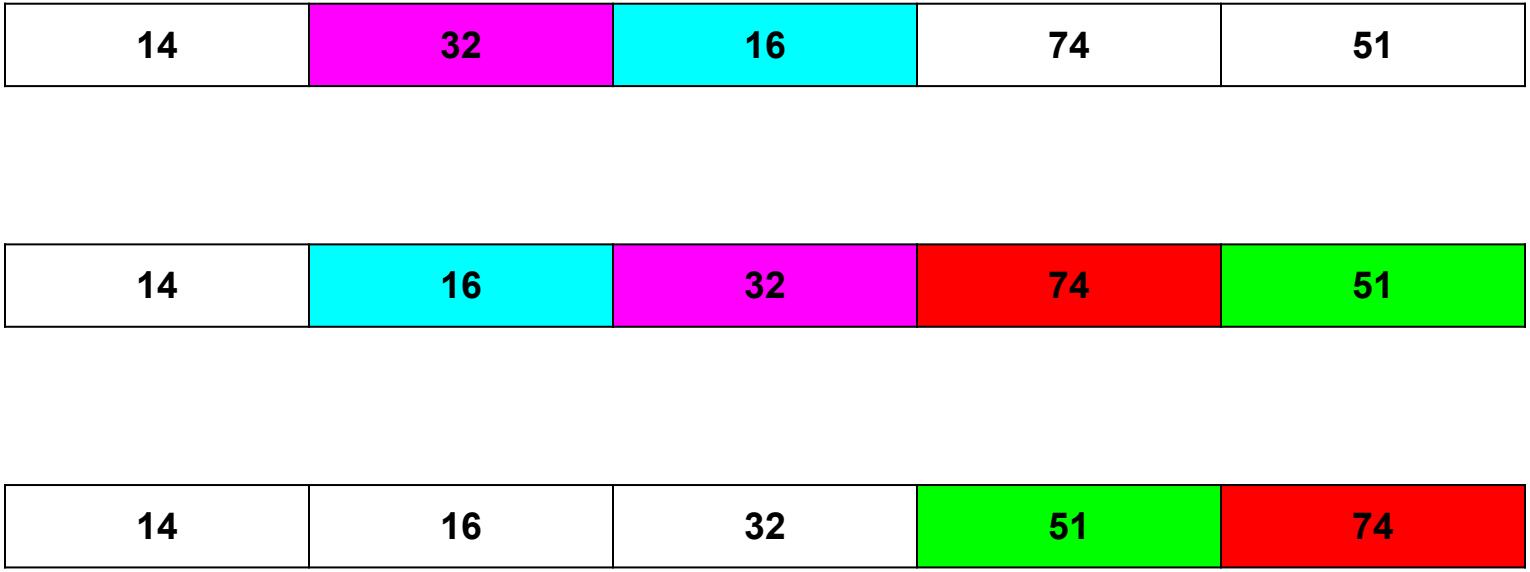
VISUALIZACIÓN COCKTAIL SORT







VISUALIZACIÓN GNOME SORT



5. CÁLCULO DE TIEMPO DE EJECUCIÓN

Nuestro siguiente paso a seguir fue calcular el tiempo de ejecución de los algoritmos anteriores.

Calculamos el tiempo de ejecución de:

Vectores :

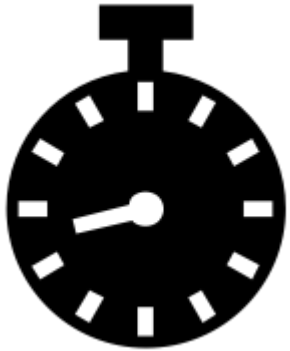
- Ordenados de forma ascendente
- Desordenados y aleatorios
- Ordenados de forma descendente

Creando, en cada uno de los casos, vectores de 5, 10, 50, 100 y 1000 elementos.

FUNCIÓN TIEMPO

Para la realización del paso anterior, requerimos la siguiente función, que nos permite calcular el tiempo de ejecución de cada algoritmo:

```
double gettime(){
 struct timeval tp;
 gettimeofday(&tp, NULL);
 return tp.tv_sec + tp.tv_usec/(double)1.0e6;
}
```



6. RECOPILACIÓN DE DATOS Y PROMEDIO

Realizamos una serie de ensayos para cada algoritmo, con cada uno de los vectores mencionados anteriormente, y, después, promediamos el tiempo en cada caso. Obteniendo así el **tiempo promedio de ejecución de cada algoritmo,** en cada caso específico.

7. GRAFICACIÓN DE LOS RESULTADOS

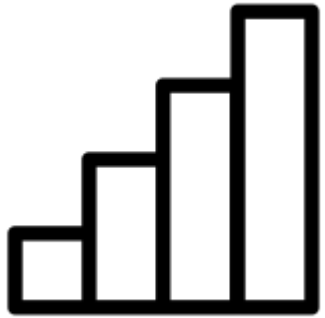
Tras haber obtenido los tiempos de ejecución de cada algoritmo en todos los casos seleccionados, realizamos gráficas que nos permiten analizar cuál de ellos es más eficiente.

Realizamos, en total, quince gráficas, que combinan los 5 tamaños posibles de vectores (5, 10, 50, 100, 500 elementos) con las 3 posibles formas de organización de los datos de los mismos (ascendente, descendente o aleatorios).

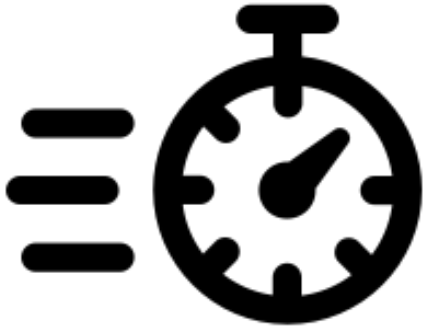
Dichas gráficas comparan tiempo de ejecución VS algoritmo de ordenamiento.

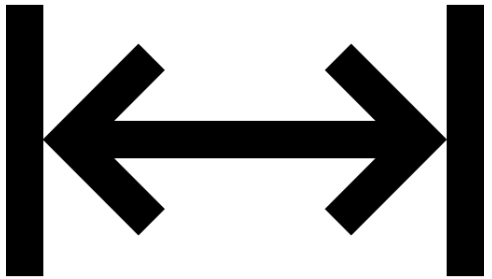
CÓDIGO PARA GRAFICAR

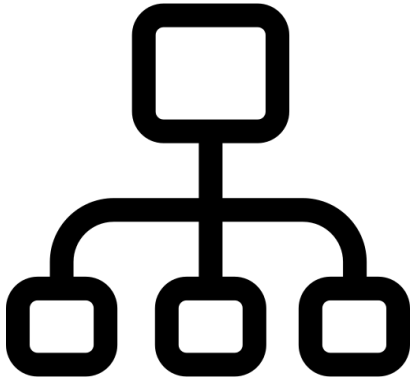
Para realizar las gráficas, utilizamos la librería de python ‘matplotlib’, realizando un código que importa los tiempos obtenidos, guardados en un archivo .txt.



8. ANÁLISIS DE LOS RESULTADOS

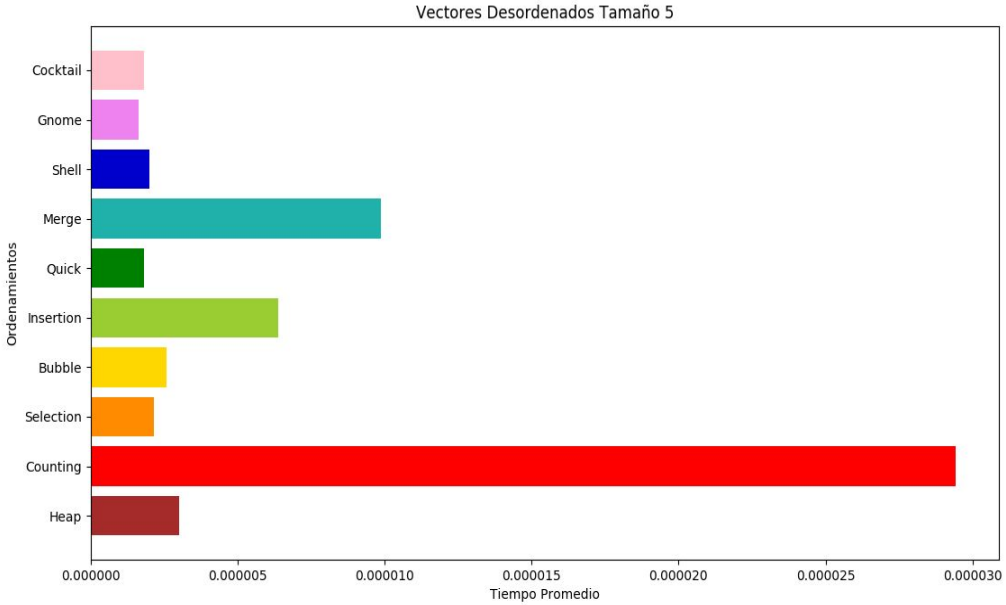


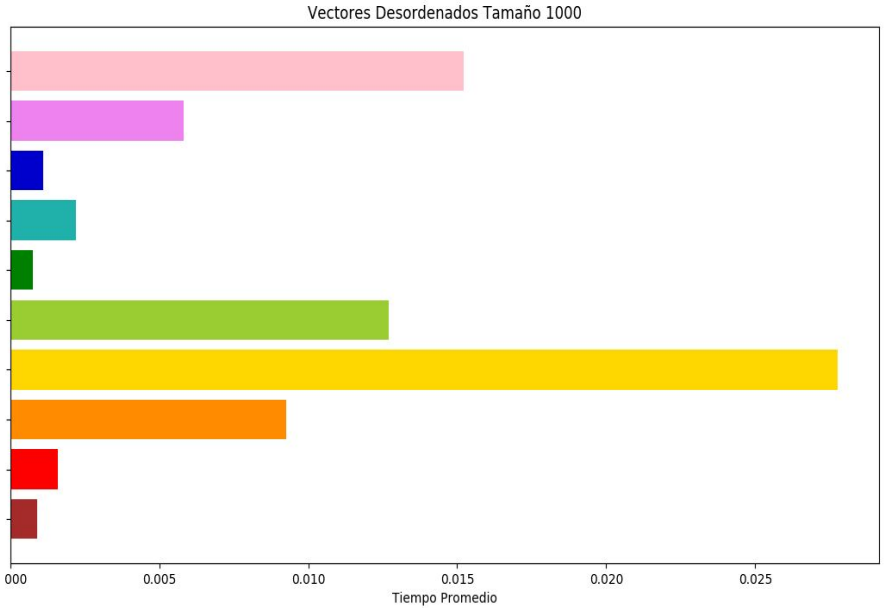




VECTORES DESORDENADOS

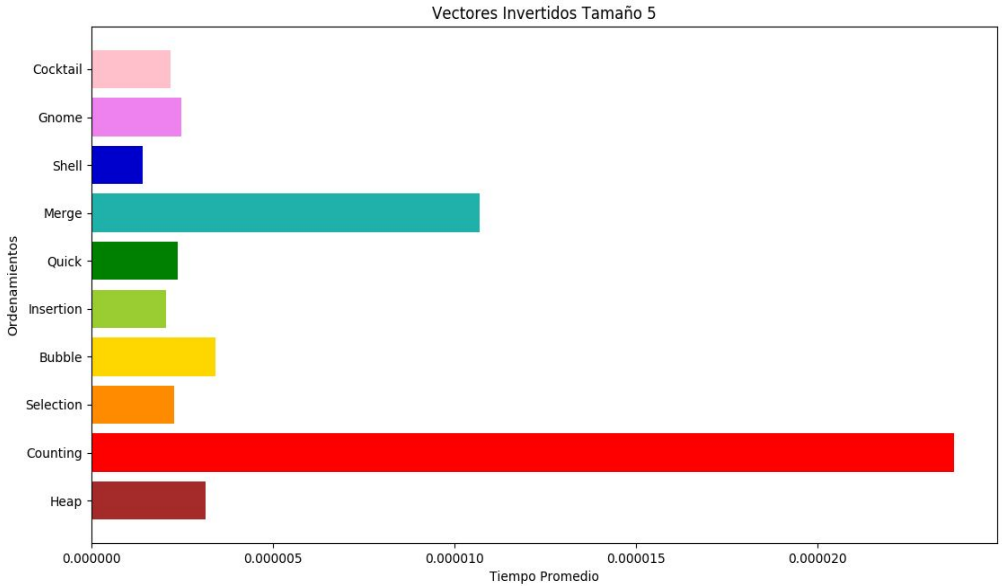
	VEC 5	VEC 10	VEC 50	VEC 100	VEC 1000
1	Gnome	Insertion	Shell	Quick	Quick
2	Quick	Quick	Quick	Shell	Heap
3	Cocktail	Shell	Heap	Heap	Shell
4	Shell	Gnome	Insertion	Counting	Counting
5	Selection	Selection	Gnome	Merge	Merge
6	Bubble	Heap	Selection	Insertion	Gnome
7	Heap	Cocktail	Cocktail	Gnome	Selection
8	Insertion	Bubble	Merge	Selection	Insertion
9	Merge	Merge	Counting	Cocktail	Cocktail
10	Counting	Counting	Bubble	Bubble	Bubble

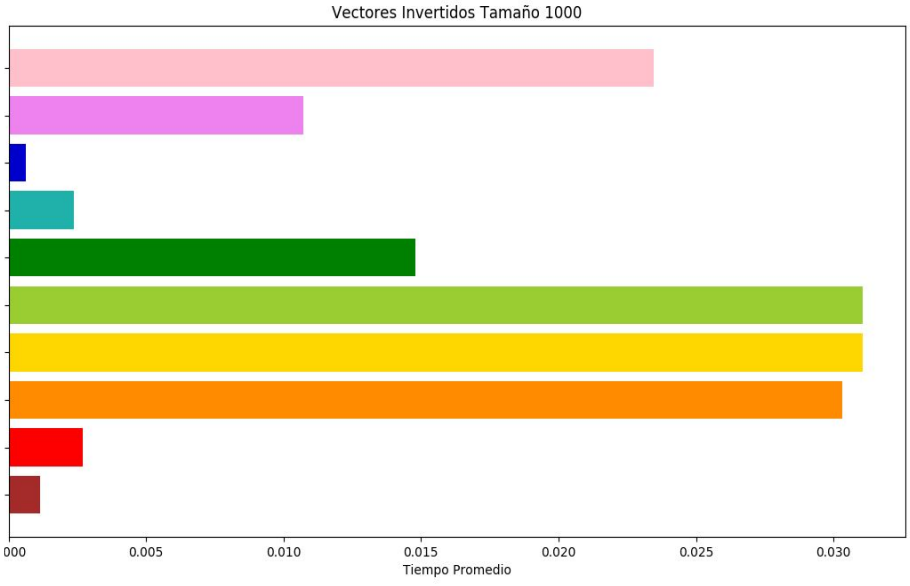

Vectores Desordenados Tamaño 5
Cocktail
Gnome
Shell
Merge
Quick
Insertion
Bubble
Selection
Counting
Heap
Ordenamientos
0.000000
0.000005
0.000010
0.000015
0.000020
0.000025
0.000030
Tiempo Promedio


Vectores Desordenados Tamaño 1000
000
0.005
0.010
0.015
0.020
0.025
Tiempo Promedio

VECTORES INVERTIDOS

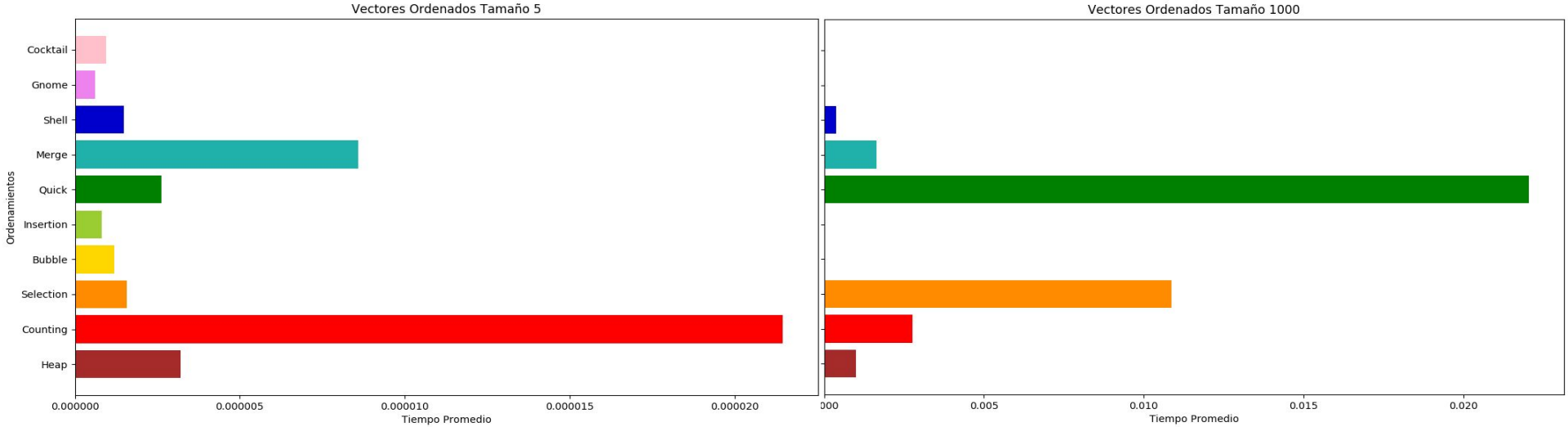
	VEC 5	VEC 10	VEC 50	VEC 100	VEC 1000
1	Shell	Shell	Shell	Shell	Shell
2	Insertion	Quick	Heap	Heap	Heap
3	Cocktail	Selection	Selection	Selection	Merge
4	Selection	Insertion	Quick	Merge	Counting
5	Quick	Gnome	Insertion	Quick	Gnome
6	Gnome	Heap	Merge	Counting	Quick
7	Heap	Cocktail	Gnome	Insertion	Cocktail
8	Bubble	Bubble	Cocktail	Gnome	Selection
9	Merge	Merge	Counting	Cocktail	Bubble
10	Counting	Counting	Bubble	Bubble	Insertion


Vectores Invertidos Tamaño 5
Cocktail
Gnome
Shell
Merge
Quick
Insertion
Bubble
Selection
Counting
Heap
Ordenamientos
0.000000
0.000005
0.000010
0.000015
0.000020
Tiempo Promedio


Vectores Invertidos Tamaño 1000
000
0.005
0.010
0.015
0.020
0.025
0.030
Tiempo Promedio

VECTORES ORDENADOS

	VEC 5	VEC 10	VEC 50	VEC 100	VEC 1000
1	Gnome	Gnome	Gnome	Gnome	Gnome
2	Insertion	Cocktail	Cocktail	Cocktail	Cocktail
3	Cocktail	Insertion	Bubble	Insertion	Bubble
4	Bubble	Bubble	Insertion	Bubble	Insertion
5	Shell	Shell	Shell	Shell	Shell
6	Selection	Selection	Heap	Heap	Heap
7	Quick	Heap	Selection	Merge	Merge
8	Heap	Quick	Merge	Selection	Counting
9	Merge	Merge	Quick	Counting	Selection
10	Counting	Counting	Counting	Quick	Quick


Vectores Ordenados Tamaño 5
Vectores Ordenados Tamaño 1000
Cocktail
Gnome
Shell
Merge
Quick
Insertion
Bubble
Selection
Counting
Heap
Ordenamientos
0.000000
0.000005
0.000010
0.000015
0.000020
000
0.005
0.010
0.015
0.020
Tiempo Promedio
Tiempo Promedio

9. CONCLUSIONES

1. No hay un algoritmo de ordenamiento en específico que se destaque más que otros.
2. Depende de la necesidad que se está buscando para determinar qué algoritmo es el más eficiente.
3. Según nuestros datos, los mejores algoritmos de ordenamientos para:
   a. Arreglos desordenados: Quick Sort
   b. Arreglos invertidos: Shell Sort.
   c. Arreglos ordenados: Gnome Sort.