Práctica 1 - ALGC

1. **¿Qué problemas introduce en una ordenación comparar distancias a un punto determinado en vez de directamente valores enteros? ¿Qué ventajas e inconvenientes tienen las alternativas que usan menos o más memoria y por qué? ¿Qué coste/complejidad tiene cada alternativa en tiempo y memoria?**

```python
def partition(A, lo, hi):
    pivot = A[hi]
    i = lo

    for j in range(lo,hi):
        if pivot>A[j]:

            A[i], A[j] = A[j], A[i]
            i = i+1

    A[i], A[hi] =A[hi], A[i]

    return i
```

1-Valores enteros

```python
def partition(A, lo, hi, ref):
    pivot = A[hi]
    i = lo

    for j in range(lo,hi):
        if dist(A[j],ref) < dist(pivot,ref):

            A[i], A[j] = A[j], A[i]
            i = i+1

    A[i], A[hi] =A[hi], A[i]

    return i
```

2-Distancia entre puntos

Al introducir la comparación de distancias añadimos dos operaciones: el cálculo de la distancia desde el punto al punto de referencia y el cálculo de la distancia desde el punto en el índice del pivote al punto de referencia.

En cada iteración del bucle for se deben calcular 2 distancias, por lo que el coste aumenta en 2n.

Comparando ambos programas con vectores del mismo tamaño, el tiempo de ejecución es similar.

2. **¿Cómo has eliminado la recursión final en el algoritmo de quicksort? ¿Cómo has introducido la inserción directa en quicksort?**

--Introducción de la inserción directa en Quicksort:

Cuando la longitud del vector a ordenar es menor o igual que el límite c, en vez de aplicar el algoritmo Quicksort, se emplea el algoritmo de inserción directa.

```python
#ALGORITMO QUICKSORT MODIFICADO

def quicksort(A, lo, hi,ref, c):

    if(len(A)<=c): #El problema de entrada es de menor o igual tamaño que c
        #print("--Inserción directa")

        insercion(A,ref)
        return

    if lo >= hi or lo < 0:
        return

    p = partition(A, lo, hi,ref)

    #Ordenar las dos particiones
    quicksort(A, lo, p-1,ref,c)
    quicksort(A, p+1, hi,ref,c)
```

Con un *if*, si el vector cumple la condición, ejecutamos dicho algoritmo, el cual devuelve el vector ordenado.

En cambio, si no cumple la condición obtenemos el pivote y ejecutamos de forma recursiva el algoritmo Quicksort.

--Eliminar la recursión final

```
def quicksort_m(A, lo, hi,ref):

    if lo >= hi or lo < 0:
        return

    p = partition(A, lo, hi,ref)

    #Ordenar las dos particiones
    quicksort_m(A, lo, p-1,ref)

    #quicksort(A, p+1, hi) Se elimina

    sinOrden = []
    lo = p+1
    sinOrden.append((lo,hi)) #Indice final e inicial de la lista que queda sin ordenar
    while len(sinOrden)>0: #Mientras la lista este llena, quedan partes por ordenar
        lo, hi = sinOrden.pop() #Coge el último elemento de la lista

        pivot = partition(A, lo, hi,ref)

        #Metemos la parte izquierda sin ordenar
        if pivot > lo +1:
            sinOrden.append((lo, pivot -1))
        #Y tambien la parte derecha del pivote
        if pivot < hi-1:
            sinOrden.append((pivot+1, hi))
```

La primera recursión se mantiene tal y como está.

La segunda la elimino y la sustituyo de la siguiente manera:

Creo un array vacío en el que insertaré los índices inferior y superior de las partes de la lista que están sin ordenar. Primero se introduce los índices superior e inferior de la parte derecha del array que es la que falta de ordenar (ya que la parte izquierda, tomando como referencia el pivote, está siendo ordenada con Quicksort).

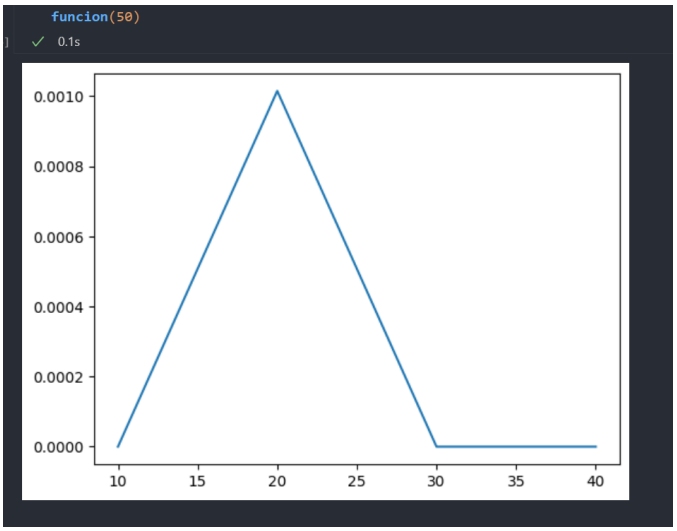
Mientras este array (sinOrden) esté lleno, quedarán partes desordenadas, por lo que debe seguir ejecutándose el programa.

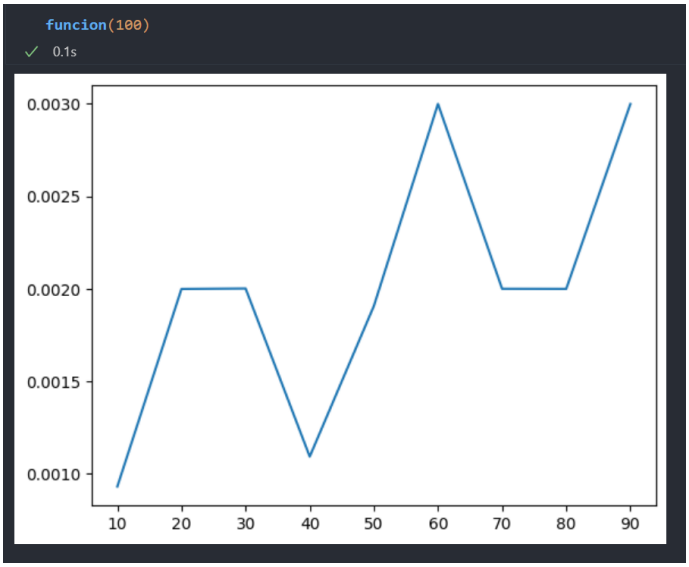
Utilizamos el algoritmo de partición de Lomuto para obtener un nuevo pivote, y añadimos a la pila las partes del array que quedan por ordenar.

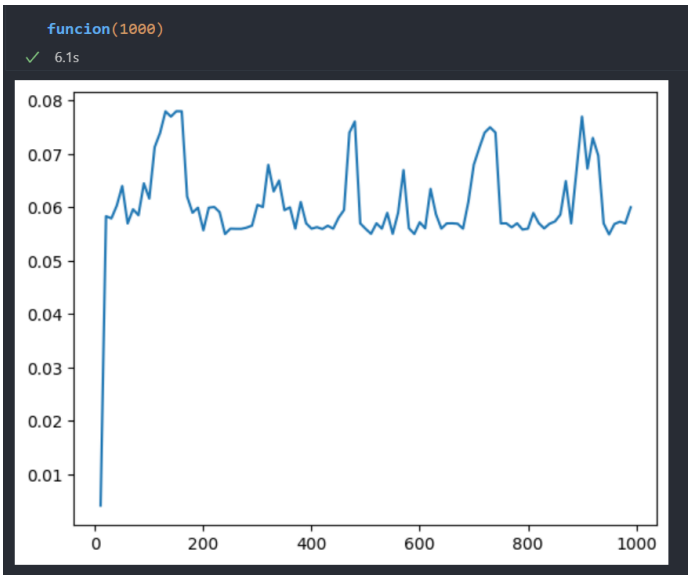
3. **¿Cómo has diseñado los experimentos para comprobar la corrección de los programas y para medir qué versión es mejor o peor? ¿Qué tamaños de problema has usado y que tiempos aproximados has obtenido?**

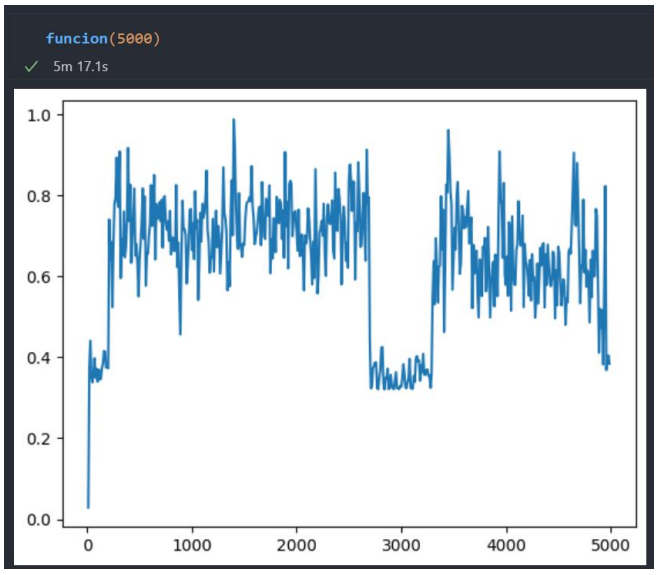
He trabajado con tiempos muy inferiores al segundo, lo que me ha dado bastantes problemas a la hora de representar los datos, pues la diferencia de los tiempos era muy pequeña, y a veces no se apreciaba en los propios gráficos.

Para el **método 1** en el que hay que buscar una C óptima he ido probando con vectores de distintos tamaños. He construido un gráfico para cada vector. En el eje X están los distintos valores de C y en el eje Y el tiempo que se ha tardado en ordenarlos (en segundos).









He empleado tamaños de vector que oscilan entre los 50 y los 5000. Con un tamaño más elevado, el coste de ejecución era muy alto, así como el tiempo de ejecución.

Para un vector de tamaño 50 (1er gráfico), la c óptima se encuentra en 30. Sin embargo, para un vector de tamaño 100, la c con menor tiempo es igual a 40.

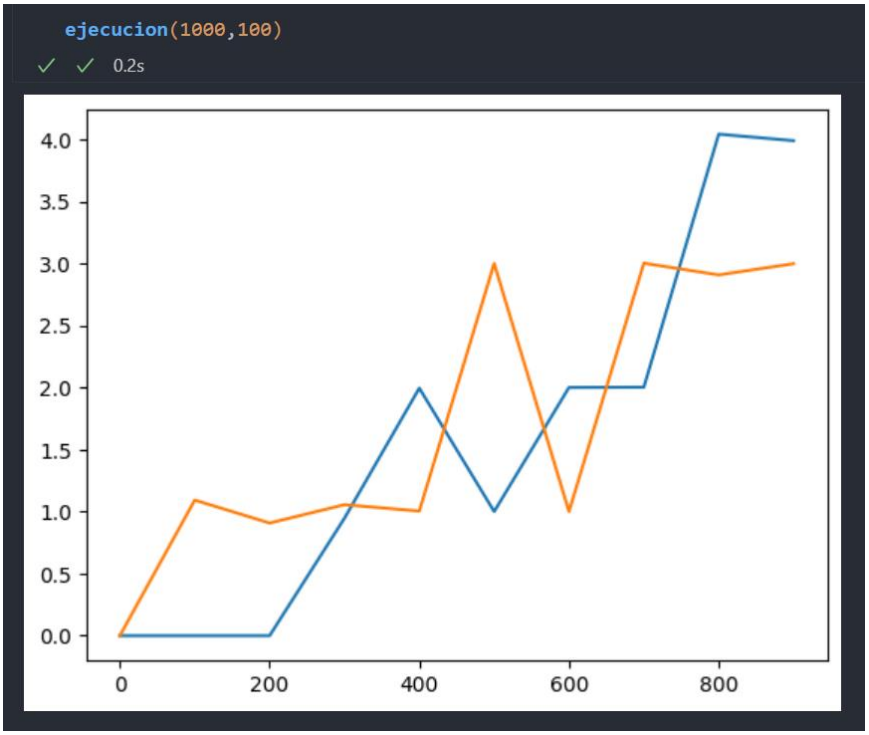
Para tamaños superiores, no he podido obtener ninguna conclusión clara. En el vector de tamaño 5000 observamos una bajada de los tiempos cuando la c ronda los 3000, lo que coincide con el 1er vector: c óptima es el 60% del tamaño del vector. Esta c del 60% del tamaño podríamos concluir que es la óptima a la hora de fijar un límite a la hora de realizar QS o ID

4. **¿Qué conclusiones has obtenido sobre los cambios en el coste al eliminar la recursión final o introducir el algoritmo de inserción directa en el quicksort? ¿Qué versiones son mejores o peores y por qué? ¿Qué lenguaje has usado, qué problemas has tenido?**

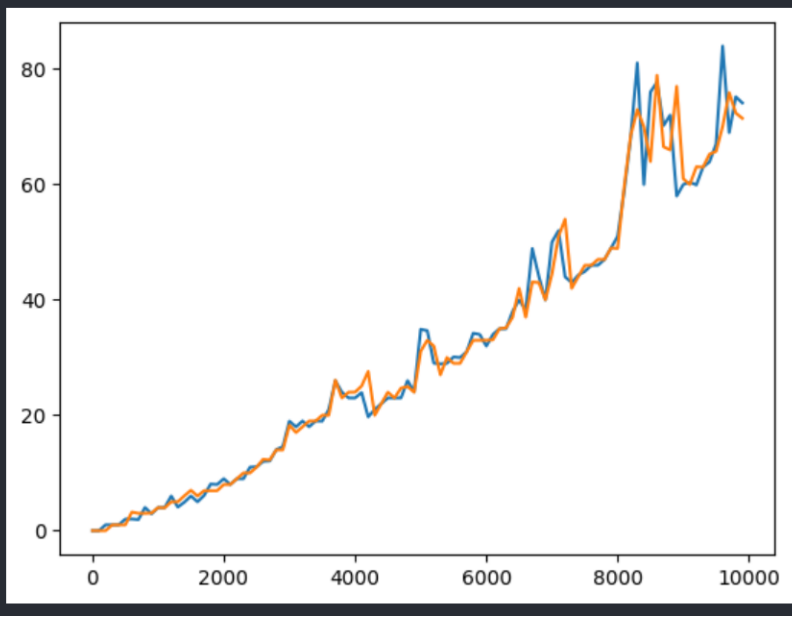
He usado como lenguaje **Python**, el cual me ha sido bastante útil a la hora de implementar arrays.

He comparado los tiempos **entre el algoritmo Quicksort y el Quicksort iterativo**, para diferentes tamaños del vector mediante gráficos en los que en el eje X aparece el tamaño del vector y en el eje Y el tiempo de ejecución en ms. La línea de color naranja corresponde a Quicksort iterativo y la azul a Quicksort con recursividad.

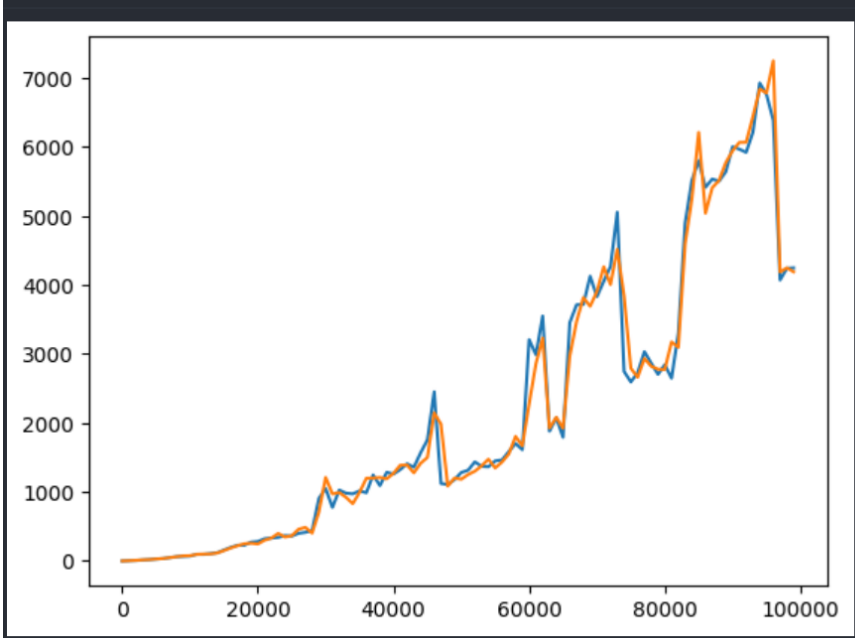
Siendo ejecución(n,k), n: tamaño máximo del vector y k: saltos de los tamaños del vector obtengo los siguientes gráficos:



```
ejecucion(10000,100)
✓ 6.9s
```

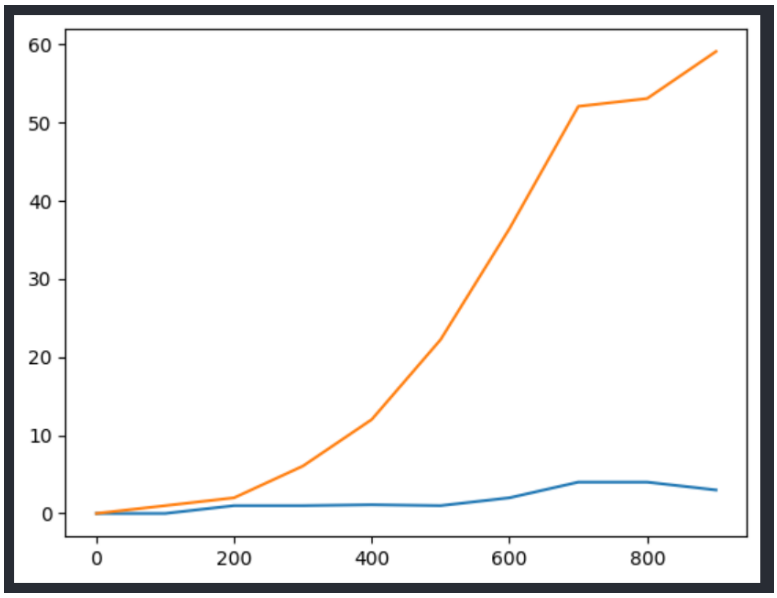



```
ejecucion(100000,1000)
✓ 7m 16.2s
```



Se puede apreciar que el comportamiento de ambos métodos (el iterativo y el recursivo) es similar, por lo que la diferencia en costes y tiempo no es muy significativa entre que Quicksort sea completamente recursivo o que incluya una parte iterativa.

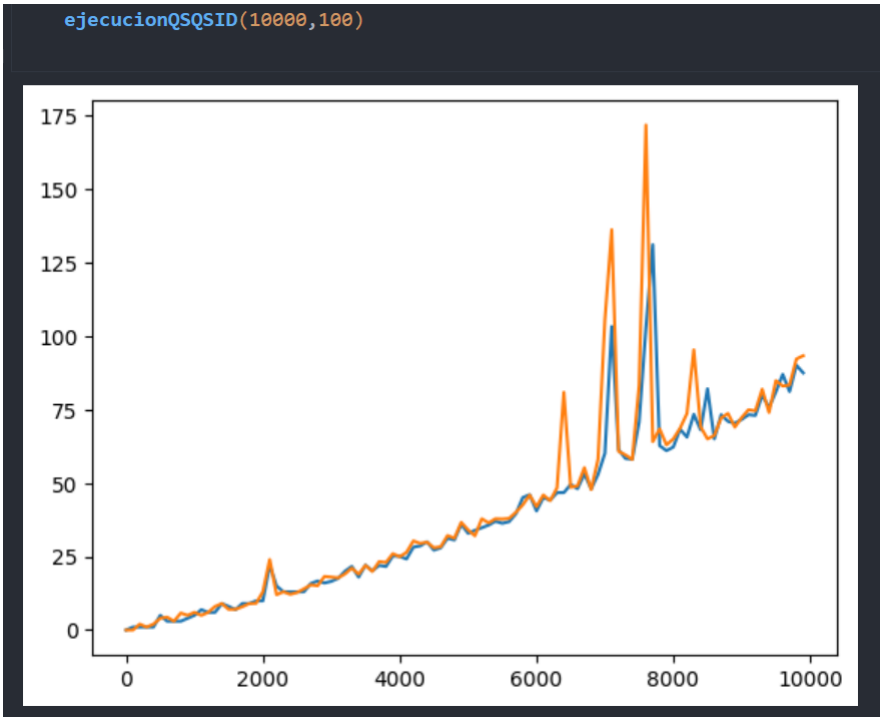
También he comparado el funcionamiento del **algoritmo Quicksort frente al de Inserción directa** de la misma manera, siendo la línea azul Quicksort y la naranja Inserción directa:



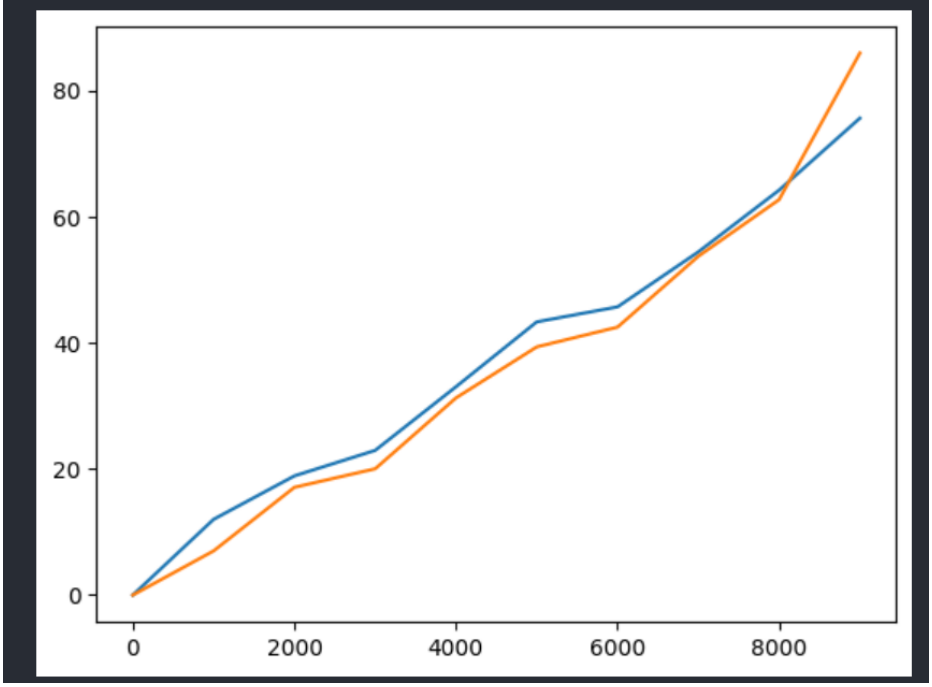
De este gráfico podemos concluir que a medida que el tamaño del vector a ordenar aumenta, el tiempo del algoritmo de Inserción directa también, mientras que el tiempo que dedica Quicksort prácticamente no varía con el tamaño del vector.

Por tanto, se concluye que con vectores de gran tamaño conviene usar Quicksort y que con vectores de un tamaño inferior a 100 es un poco más indiferente.

Además, he comparado los tiempos de ejecución de **Quicksort con Quicksort modificado** (Quicksort + Inserción Directa). En el eje X se indica el tamaño del vector y en el eje Y el tiempo empleado en milisegundos. La línea naranja corresponde al algoritmo Quicksort con Inserción Directa y la azul al algoritmo Quicksort.



```
ejecucionQSQSID(10000,1000)
```



Los tiempos son muy similares para ambos métodos, tanto en vectores de pequeño como gran tamaño, por lo que el uso de los dos métodos debería ser indiferente.