Análisis de Algoritmos 2025/2026

Práctica 1

Shaofan Xu, Alejandro Zheng, (120/127)

Código	Gráficas	Memoria	Total

1. Introducción.

En está práctica hemos implementado algoritmo de ordenación como InsertSort y BubbleSort, y así análisis de eficiencia de algoritmo. El objetivo es comparar el análisis teórico con respecto al rendimiento real de los algoritmos, midiendo tanto el tiempo de ejecución como el número de las operaciones básicas.

2. Objetivos

2.1 Apartado 1

El objetivo es implementar una función, random_num, capaz de generar números enteros aleatorios dentro de un intervalo definido (inf, sup). Con la distribución de los números generados sea equiprobable.

2.2 Apartado 2

Se busca implementar la función generate_perm, que utilizará la rutina del apartado anterior para generar una única permutación aleatoria de N elementos, siguiendo el algoritmo de Fisher-Yates (descrito en el pseudocódigo).

2.3 Apartado 3

El objetivo es extender la funcionalidad anterior mediante la creación de generate_permutations. Esta función generará y almacenará en memoria un conjunto de n_perms permutaciones de tamaño N, que servirán como datos de entrada para los algoritmos de ordenación.

2.4 Apartado 4

En este apartado se implementará el algoritmo de ordenación por inserción, InsertSort. La función no solo deberá ordenar correctamente un subconjunto de un array (ip, iu), sino que también deberá contar y devolver el número de veces que se ejecuta su operación básica.

2.5 Apartado 5

El objetivo es construir un marco completo para la medición empírica del rendimiento de un algoritmo de ordenación. Esto implica:

- Crear la función average_sorting_time para calcular el tiempo medio y las estadísticas de la operación básica (promedio, mínimo, máximo) para un tamaño de entrada N.
- Implementar generate_sorting_times y save_time_table para automatizar la ejecución del análisis sobre un rango de tamaños de entrada y guardar los resultados de manera estructurada en un fichero.

2.6 Apartado 6

Finalmente, el objetivo es implementar el algoritmo BubbleSort y utilizar el sistema de medición desarrollado en el apartado 5 para realizar un análisis comparativo entre InsertSort y BubbleSort. Se compararán los tiempos de ejecución y las estadísticas de sus operaciones básicas (caso promedio, mejor y peor) para evaluar empíricamente sus diferencias de eficiencia.

3. Herramientas y metodología

Para el desarrollo de esta práctica se ha utilizado un entorno de desarrollo basado en Linux (Ubuntu 24.04.3), aprovechando las herramientas de la línea de comandos que ofrece este sistema.

Las herramientas principales empleadas han sido:

- **Editor de Código:** Visual Studio Code.
- **Compilador:** gcc, utilizando los flags -Wall para activar todas las advertencias, -pedantic para generar la información necesaria para la depuración, -ansi para usar el standard C89. Y, además -O3 para optimizar el programa.
- **Análisis de Memoria:** Valgrind, una herramienta fundamental para detectar fugas de memoria y accesos a memoria inválidos.
- **Automatización:** make, mediante un fichero Makefile, para automatizar el proceso de compilación y ejecución de los distintos ejercicios.
- **Visualización de Datos:** Gnuplot, para generar las gráficas a partir de los ficheros de resultados y poder realizar un análisis visual comparativo del rendimiento de los algoritmos.

3.1 Apartado 1:

```
1. int random_num(int inf, int sup)
```

Metodología y solución adoptada del apartado 1: Para evitar problema de operador **(%)**, ya que no genera números aleatorios equiprobables. La solución de implementada se basa en la característica de la función **rand()**, donde devuelve un numero entre **0** y **RAND_MAX**, lo cual **rand()/(RAND_MAX+1)** nos devuelve un número que pertenece a **[0,1)**, multiplicando a **(sup-inf+1)** nos devuelve un número perteneciente a **[0,sup-inf]**, luego al sumar inf nos devuelve números entre **[inf,sup]**. Por lo tanto, hacemos que la función devuelve el valor **inf+(sup-inf+1)*(double)(rand()/(RAND_MAX+1))**.

Pruebas: Para comprobar hemos generado números entre 1 y 2, así generando 10 números.

3.2 Apartado 2:

```
1. int* generate_perm(int N)
```

Metodología y solución adoptada del apartado 2: Se implementó directamente el algoritmo de Fisher-Yates tal como se describía en el pseudocódigo. La función

primero reserva memoria para un array de N enteros, lo inicializa con los valores de 1 a N, y luego recorre el array intercambiando cada elemento con otro elegido al azar entre su propia posición y el final del array.

Pruebas: Hemos hecho prueba con 5 permutación de tamaño 5.

3.3 Apartado 3:

```
1. int** generate_permutations(int n_perms, int N)
```

Metodología y solución adoptada del apartado 3: Se implementó una función que primero reserva memoria para un array de punteros (int **). A continuación, dentro de un bucle que se repite n_perms veces, se llama a la función generate_perm del apartado anterior y se almacena el puntero devuelto en el array. En el caso de que falla en la reserva de la memoria, se libera todas las memorias reservadas anteriormente.

Pruebas: Al igual que apartado anterior, hemos hecho prueba con 5 permutación de tamaño 5.

3.4 Apartado 4:

```
1. int InsertSort(int* array, int ip, int iu)
```

Metodología y solución adoptada del apartado 4: Se realizó una implementación del algoritmo de **InsertSort**. El código consiste en un bucle externo que recorre el array desde el segundo elemento, y un bucle while interno que desplaza los elementos mayores hacia la derecha para hacer sitio al elemento actual. La Operación Básica (OB) contamos por cada comparación del bucle while.

Pruebas: Hemos hecho prueba con una permutación de tamaño 5.

3.5 Apartado 5:

```
1. short average_sorting_time(pfunc_sort metodo, int n_perms,int N, PTIME_AA ptime)
2. short generate_sorting_times(pfunc_sort method, char* file, int num_min, int num_max, int
incr, int n_perms)
3. short save_time_table(char* file, PTIME_AA ptime, int n_times)
```

Metodología y solución adoptada del apartado 5: Se realizó implementación de varias funciones para medir la eficiencia de los algoritmos de ordenación. Primero, **average_sorting_time**, genera las permutaciones, ordenan las permutaciones con el algoritmo de ordenación, y con la función clock() calculamos la media de tiempo que ejecuta el algoritmo sobre todas ellas, y calcula los promedios y extremos (mínimo/máximo) de las OBs.

Posteriormente, **generate_sorting_times** llamando a la función anterior en un bucle para un rango de tamaños (de num_min a num_max) y almacenando cada resultado en un array de estructuras TIME_AA.

Finalmente, **save_time_table** recorre el array de resultados y los escribe en un fichero de texto para ser procesado por otras herramientas como Gnuplot.

Pruebas: Para hacer media hemos probado con hacer 10 permutación para cada tamaño de 10 hasta 10000, incrementado de 10 en 10.

3.6 Apartado 6:

```
1. int BubbleSort(int* array, int ip, int iu)
```

Metodología y solución adoptada del apartado 6: Primero, se implementó el algoritmo **BubbleSort** que utiliza una bandera (flag). La OB se contó en cada comparación. A continuación, se reutilizó el trabajo del apartado 5, ejecutándolo una vez para InsertSort y otra para BubbleSort, generando dos ficheros de datos.

Pruebas: Para hacer media hemos probado con hacer 10 permutación para cada tamaño de 10 hasta 10000, incrementado de 10 en 10.

4. Código fuente

4.1 Apartado 1

```
1. int random_num(int inf, int sup)
2. {
3.   /* Return equiprobable random numbers */
4.   return inf+(sup-inf+1)*(double)(rand()/((double)RAND_MAX+1.0));
5. }
```

4.2 Apartado 2

```
 1. void swap(int *a,int *b)
 2. {
 3.   int number;
 4.   number=*a;
 5.   *a=*b;
 6.   *b=number;
 7. }
 8. int* generate_perm(int N)
 9. {
10.   int *perm=NULL;
11.   int i;
12.
13.   /* Verified that N its not zero or negative number*/
14.   assert(N>0);
15.
16.   perm=malloc(N*sizeof(perm[0]));
17.   if(perm==NULL) return NULL;
18.
19.   for(i=0;i<N;i++) perm[i]= i+1;
20.
21.   for(i=0;i<N;i++)
22.   {
23.     /* Switch with the element at random position */
24.     swap(&perm[i],&perm[random_num(i,N-1)]);
25.   }
26.
27.   return perm;
28. }
```

4.3 Apartado 3

```
1. int** generate_permutations(int n_perms, int N)
2. {
3.
4.   int **perms=NULL;
5.   int i,j;
6.
7.   /* Error comprobation*/
8.   assert(n_perms>0);
9.   assert(N>0);
10.
11.  /*Reserve memory for array*/
12.  perms=malloc(n_perms*sizeof(perms[0]));
13.  if(perms==NULL) return NULL;
14.
15.  for(i=0;i<n_perms;i++)
16.  {
17.    /* Generate permutation*/
18.    perms[i]=generate_perm(N);
19.    if(perms[i]==NULL)
20.    {
21.      for(j=0;j<i;j++)
22.      {
23.        free(perms[j]);
24.      }
25.      free(perms);
26.    }
27.  }
28.
29.  return perms;
30. }
```

4.4 Apartado 4

```
1. int InsertSort(int* array, int ip, int iu)
2. {
3.
4.   int i = 0,j;
5.   int num;
6.   int ob = 0;
7.
8.   /*Error comprobation*/
9.   assert(array != NULL);
10.  assert(ip >= 0);
11.  assert (ip <= iu);
12.
13.  /*Order the array*/
14.  for (i = ip+1; i<=iu; i++)
15.  {
16.    num = array[i];
17.    j = i - 1;
18.
19.    while (j >= ip && array[j] > num)
20.    {
21.      ob++;
22.      array[j+1] = array[j];
23.      j--;
24.    }
25.
26.    array[j+1] = num;
27.  }
28.
29.  return ob;
30. }
```

4.5 Apartado 5

```
 1.  short average_sorting_time(pfunc_sort metodo, int n_perms,int N, PTIME_AA ptime)
 2. {
 3.   int i;
 4.   int** permutations=NULL;
 5.   clock_t start,end;
 6.   double mean_time;
 7.   long mean_ob=0;
 8.   int max_ob,min_ob,ob;
 9.
10.   /*Error comprobation*/
11.   assert(metodo!=NULL);
12.   assert(n_perms>0);
13.   assert(N>0);
14.   assert(ptime!=NULL);
15.
16.   /*Generate n_perms permutations of size N and verify that is not NULL*/
17.   permutations=generate_permutations(n_perms,N);
18.   if(permutations==NULL) return ERR;
19.
20.   /* Count the time used to order*/
21.   start=clock();
22.
23.   for(i=0;i<n_perms;i++)
24.   {
25.     ob=metodo(permutations[i],0,N-1);
26.     mean_ob+=ob;
27.     if(i==0)
28.     {
29.       max_ob=min_ob=ob;
30.     }
31.     max_ob=max_ob>ob?max_ob:ob;
32.     min_ob=min_ob<ob?min_ob:ob;
33.   }
34.   end=clock();
35.
36.   mean_time=(double)(end-start)/n_perms;
37.
38.
39.   /* Asignation of values to ptime */
40.   ptime->n_elems=n_perms;
41.   ptime->N=N;
42.   ptime->time=mean_time/CLOCKS_PER_SEC;
43.   ptime->average_ob=(double)(mean_ob/n_perms);
44.   ptime->min_ob=min_ob;
45.   ptime->max_ob=max_ob;
46.
47.   /* Free all memorys */
48.
49.   for(i=0;i<n_perms;i++)
50.   {
51.     free(permutations[i]);
52.   }
53.   free(permutations);
54.
55.     return OK;
56. }
```

```
 1. short generate_sorting_times(pfunc_sort method, char* file, int num_min, int num_max,
int incr, int n_perms)
 2. {
 3.
 4.   PTIME_AA array_time = NULL;
 5.   int i;
 6.   int n_times;
 7.   int N;
 8.   short result;
```

```
 9.
10.   assert(method != NULL);
11.   assert(file != NULL);
12.   assert(num_min >0);
13.   assert(num_min < num_max);
14.
15.   n_times = (num_max -num_min) / incr + 1;
16.
17.   array_time = malloc(sizeof(TIME_AA)*n_times);
18.   if(array_time == NULL)
19.   {
20.     return ERR;
21.   }
22.
23.   for(i=0,N = num_min; N <= num_max&&i<n_times; i++, N+= incr)
24.   {
25.     if (average_sorting_time(method, n_perms,N, &(array_time[i])) == ERR)
26.     {
27.       free(array_time);
28.       return ERR;
29.     }
30.   }
31.
32.   result = save_time_table(file, array_time, n_times);
33.
34.   free(array_time);
35.
36.   return result;
37.
38. }
39.
```

```
 1. short save_time_table(char* file, PTIME_AA ptime, int n_times)
 2. {
 3.   FILE *pf = NULL;
 4.   int i;
 5.
 6.   assert(file != NULL);
 7.   assert(ptime != NULL);
 8.   assert(n_times > 0);
 9.
10.   pf = fopen(file,"w");
11.   if (pf == NULL)
12.   {
13.     return ERR;
14.   }
15.
16.   fprintf(pf, "N  Time  average_ob  max_ob  min_ob \n");
17.
18.   for (i = 0; i < n_times; i++)
19.   {
20.      fprintf(pf, "%d %f %f %d %d \n", ptime[i].N, ptime[i].time, ptime[i].average_ob,
ptime[i].max_ob, ptime[i].min_ob);
21.   }
22.
23.   fclose(pf);
24.
25.   return OK;
26. }
```

4.6 Apartado 6

```
int BubbleSort(int* array, int ip, int iu)
{
  int i=iu,j;
  int ob=0;
  int flag=1;
  int num;

  /*Error comprobation*/
  assert(array!=NULL);
  assert(ip>=0);
  assert(ip<=iu);

  while(flag==1&&i>=ip+1)
  {
    flag=0;
    for(j=ip;j<i;j++)
    {
      ob++;
      if(array[j]>array[j+1]){
        num=array[j];
        array[j]=array[j+1];
        array[j+1]=num;
        flag=1;
      }
    }
    i--;
  }
  return ob;
}
```

5. Resultados, Gráficas

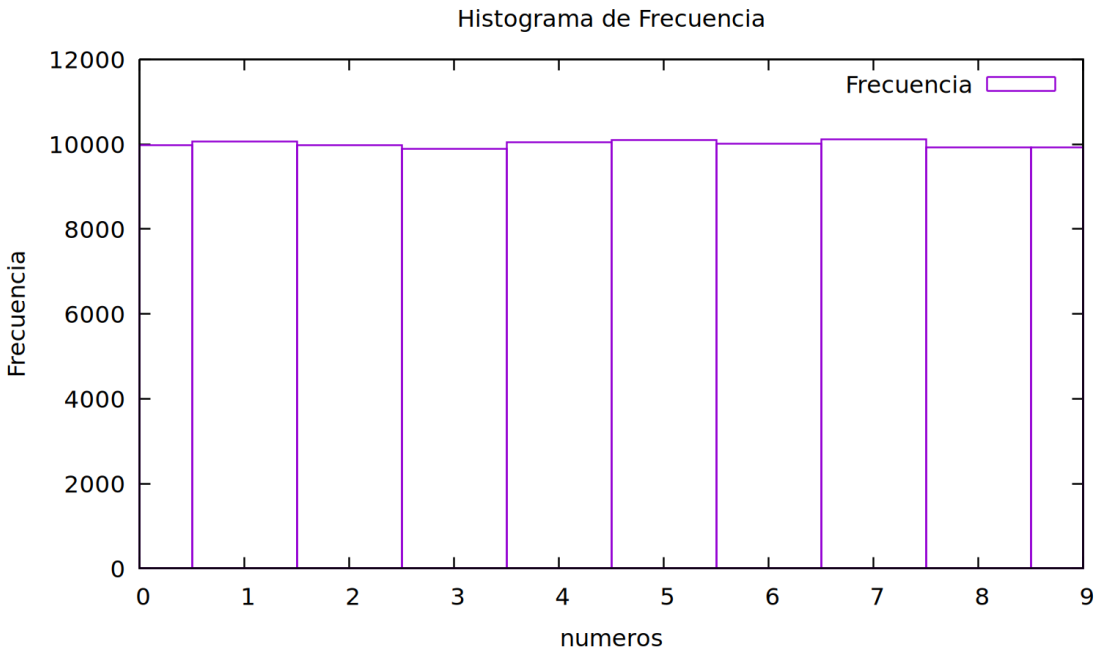
Aquí ponis los resultados obtenidos en cada apartado, incluyendo las posibles gráficas.

5.1 Apartado 1

Resultados del apartado 1.

```
Running exercise1
Practice no 1, Section 1
Done by: Shaofan Xu y Alejandro Zheng
Grupo: 120/127
2
1
1
2
1
2
2
1
1
2
```

Gráfica del histograma de números aleatorios, comentarios a la gráfica



Para ver de forma mejor que los números generados son equiprobables, hemos generado números entre 0 y 9, así generando 100000 números. Así, podemos visualizar que el programa cumple con nuestra expectativa, ya que casi todos los números aparecen las mismas veces.

5.2 Apartado 2

- Resultados del apartado 2.

```
1. Running exercise2
2. Practice number 1, section 2
3. Done by: Shaofan Xu y Alejandro Zheng
4. Grupo: 120/127
5. 5 4 3 1 2
6. 1 3 5 4 2
7. 2 1 3 5 4
8. 3 2 4 1 5
9. 2 3 5 4 1
```

5.3 Apartado 3

- Resultados del apartado 3.

```
1. Running exercise3
2. Practice number 1, section 3
3. Done by: Shaofan Xu y Alejandro Zheng
4. Grupo: 120/127
5. 4 3 5 1 2
6. 3 4 2 5 1
7. 5 4 2 3 1
8. 3 5 1 4 2
9. 1 2 5 4 3
```

5.4 Apartado 4

- Resultados del apartado 4.

```
1. Running exercise4
2. Practice number 1, section 4
3. Done by: Shaofan Xu y Alejandro Zheng
4. Grupo: 120/127
5. 1       2       3       4       5
```

5.5 Apartado 5

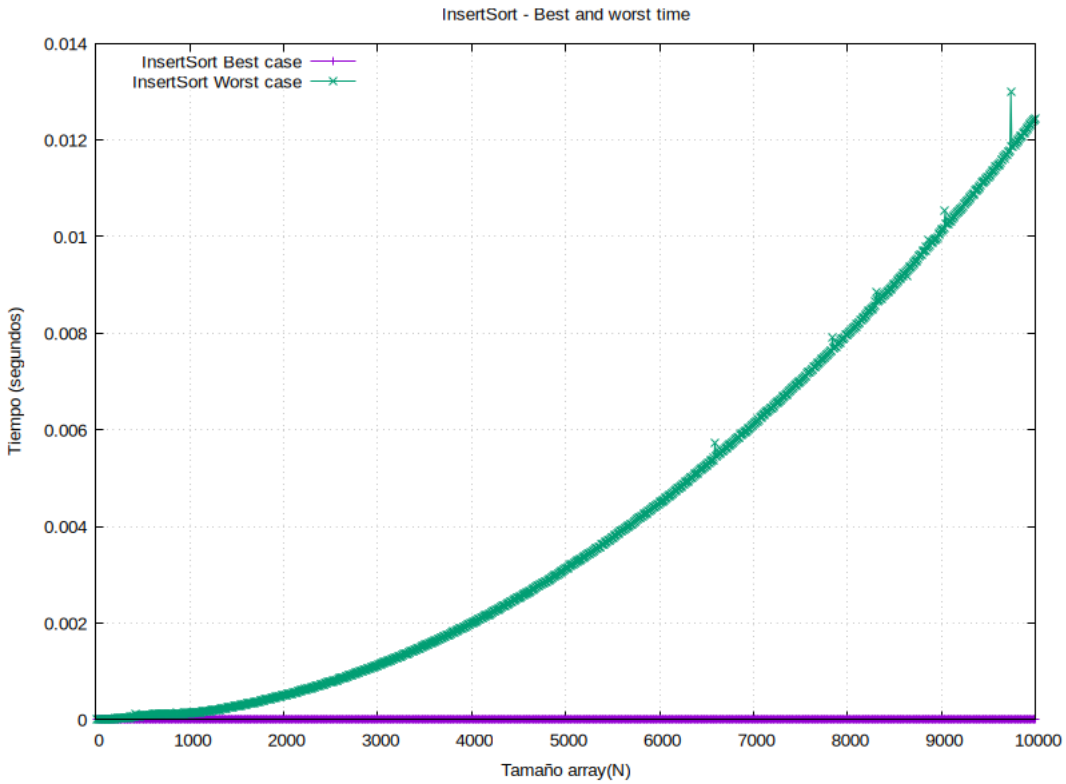
- Resultados del apartado 5.

```
1. Running exercise5
2. Practice number 1, section 5
3. Done by: Shaofan Xu y Alejandro Zheng
4. Grupo: 120/127
5. Correct output
```

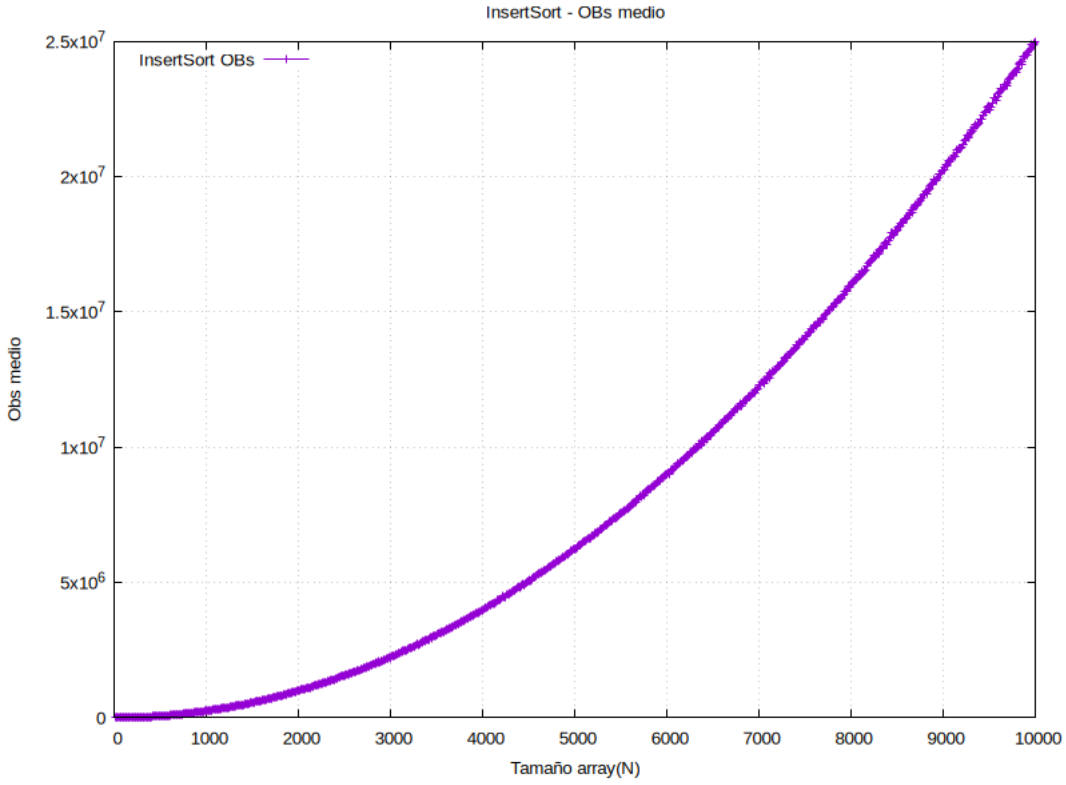
```
 1.  N  Time  average_ob  max_ob  min_ob
 2. 10 0.000000 24.000000 33 19
 3. 20 0.000000 93.000000 123 76
 4. 30 0.000000 207.000000 253 171
 5. 40 0.000001 370.000000 436 316
 6. 50 0.000001 585.000000 681 507
 7. 60 0.000001 874.000000 1100 777
 8. 70 0.000001 1221.000000 1411 994
 9. 80 0.000002 1586.000000 1673 1476
10. 90 0.000003 2075.000000 2296 1812
11. 100 0.000004 2465.000000 2804 2127
12.  …… ver exercise5.log
```

- Gráfica comparando los tiempos mejor, peor y medio en OBs para InsertSort, comentarios a la gráfica.

Para la comparación de los tiempos mejor y peor, hemos desarrollado un programa para estimar el rendimiento del algoritmo de ordenación en el caso mejor y en caso peor, midiendo su rendimiento en segundos (ver **best_worst_time.c**).

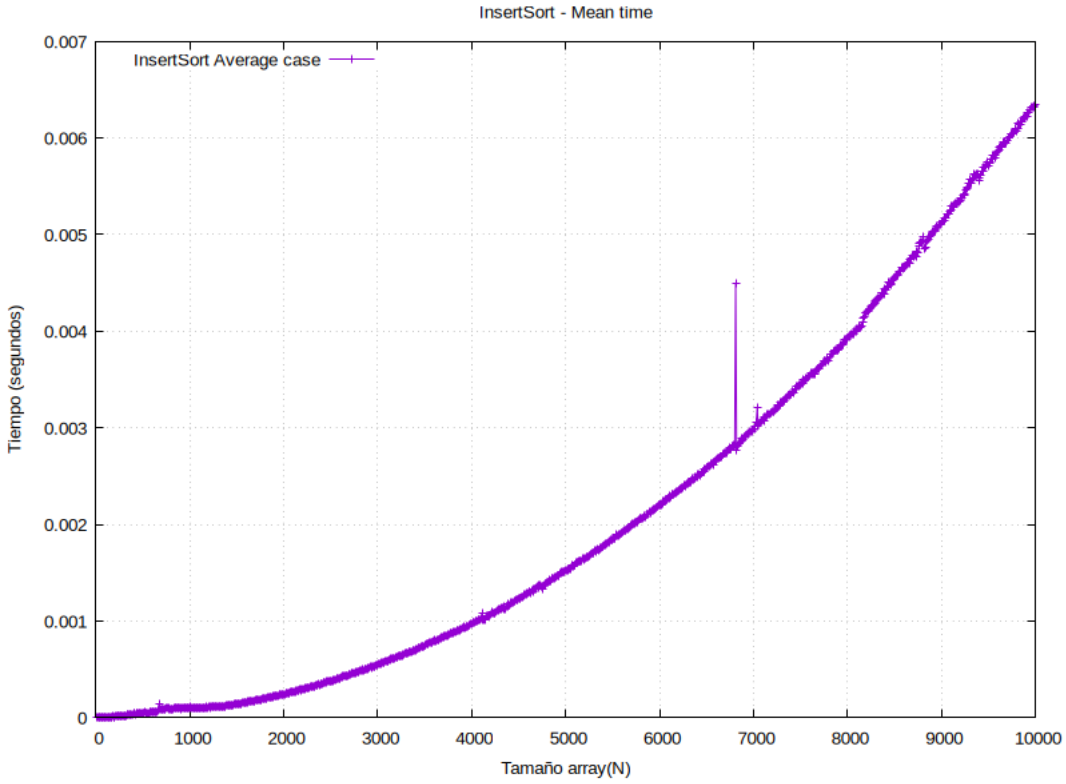


Pues la gráfica muestra una clara diferencia entre el caso mejor y peor, una muestra característica de una función de segundo grado, sin embargo, otra casi una recta constante(ya que los ordenadores son demasiado rápidos y también que la función clock() no es capaz de medir con precisión en un intervalo corto).



Pues la gráfica de la media de las OBs presenta una relación cuadrática de OBs con respecto al tamaño del array, es decir, es lo que hemos visto en la teoría.

- Gráfica con el tiempo medio de reloj para InsertSort, comentarios a la gráfica.



Al igual que el caso anterior, la gráfica de tiempo_medio-tamaño_array tiene también un comportamiento cuadrático, pues es totalmente lógico, ya que en la gráfica anterior de OBs_medio-tamaño_array también es cuadrático, y por cada operación básica que realiza el algoritmo de ordenación aporta una unidad de tiempo, así que, si la operación básica crece cuadráticamente con respecto al tamaño de array, pues también el tiempo con respecto al tamaño del array.

5.6 Apartado 6

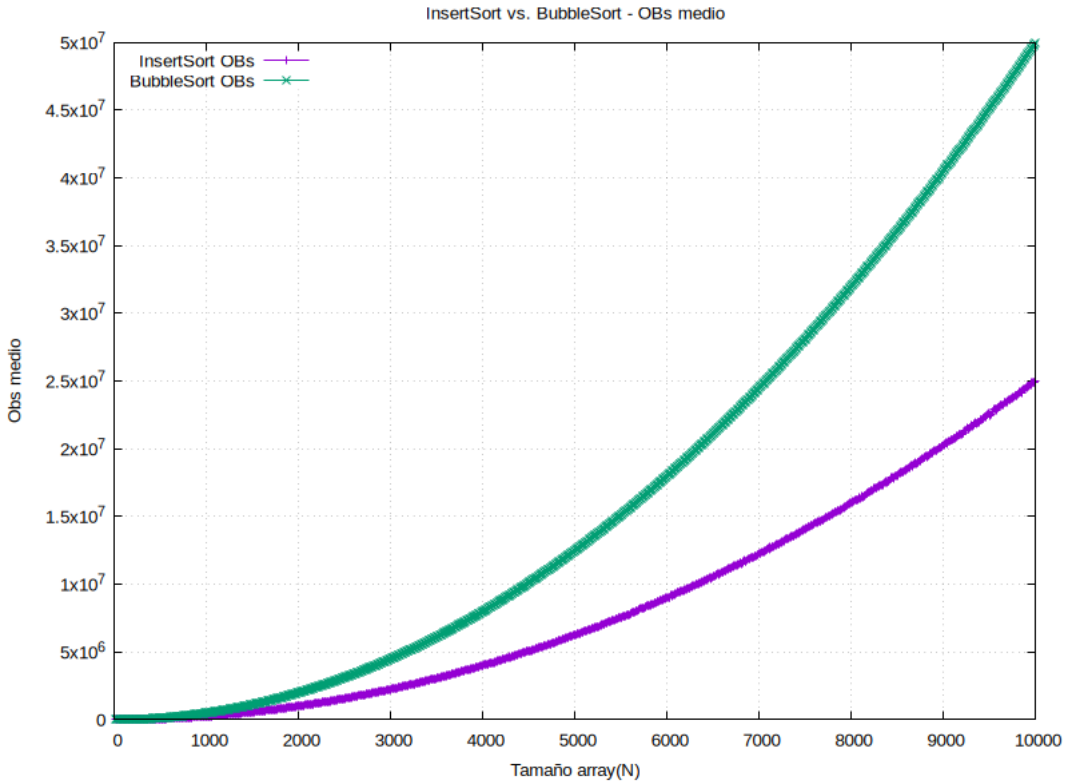
- Resultados del apartado 6.

```
1. Running exercise6
2. Practice number 1, section 6
3. Done by: Shaofan Xu y Alejandro Zheng
4. Grupo: 120/127
5. Correct output
```

```
 1. N  Time  average_ob  max_ob  min_ob
 2. 10 0.000000 41.000000 45 35
 3. 20 0.000001 181.000000 190 145
 4. 30 0.000001 421.000000 435 399
 5. 40 0.000002 754.000000 780 660
 6. 50 0.000003 1192.000000 1222 1054
 7. 60 0.000003 1721.000000 1767 1494
 8. 70 0.000004 2375.000000 2414 2279
 9. 80 0.000005 3107.000000 3160 2970
10. 90 0.000006 3957.000000 4005 3869
```

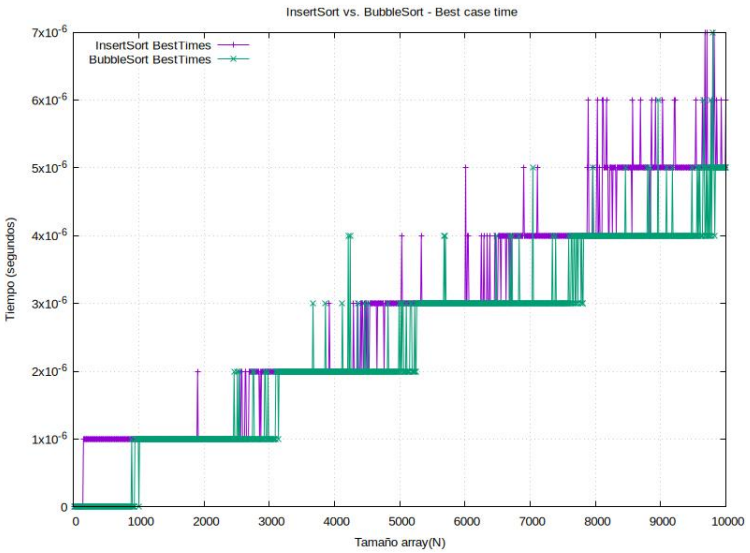
```
11. 100 0.000007 4889.000000 4944 4740
12.  …… ver exercise6.log
```

- Gráfica comparando el tiempo medio de OBs para InsertSort y BubbleSort, comentarios a la gráfica.



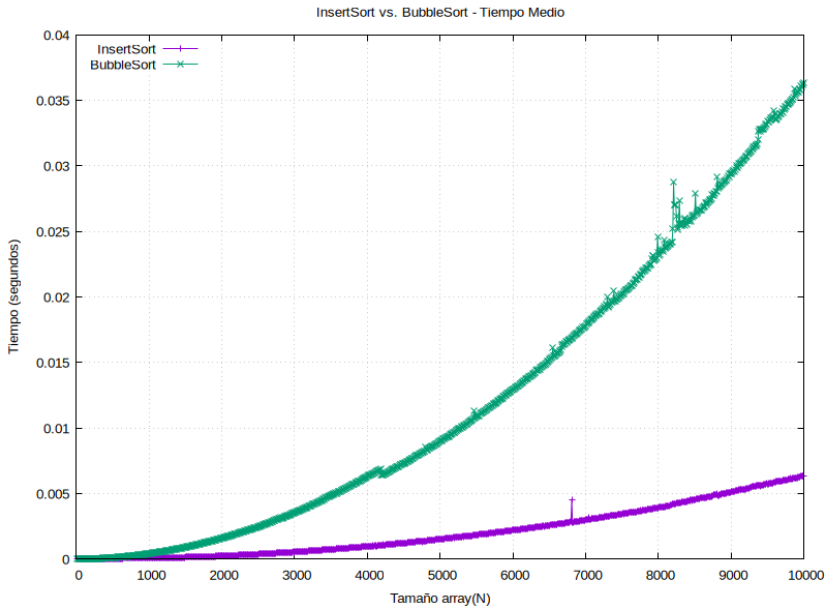
En esta gráfica que compara el rendimiento medio en OBs de InsertSort y BubbleSort, podemos ver claramente que el InsertSort crece más lento que BubbleSort, y ambas tienen un comportamiento cuadrático.

- Gráfica comparando el tiempo mejor para InsertSort y BubbleSort, comentarios a la gráfica.



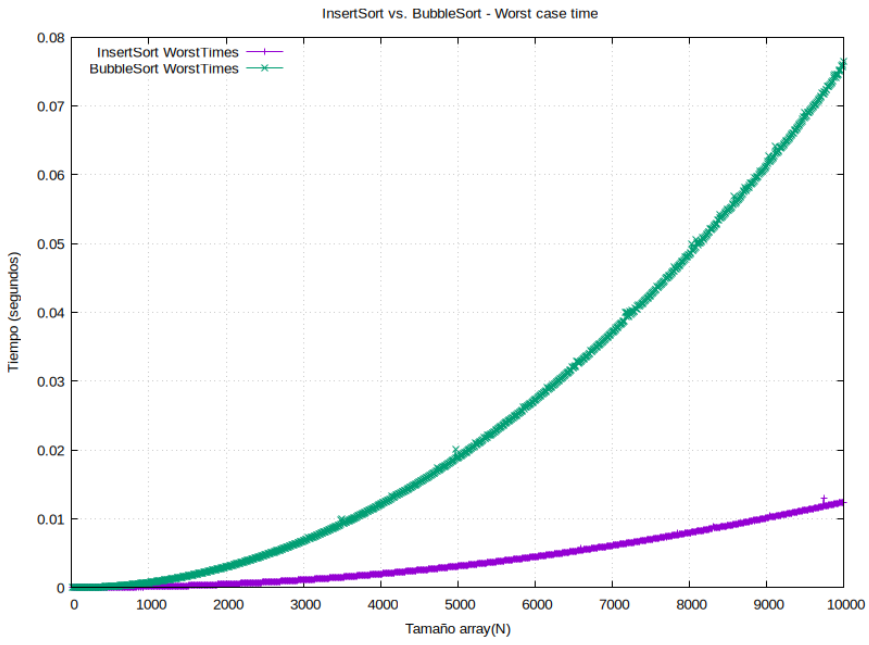
En está gráfica que vemos, aunque haya muchos saltos, pero debido a la escala, sabemos al final comporta como una recta O(N) en el caso mejor, y estos pequeños saltos son debido al que la función que usamos clock() no son precisa a pequeñas escalas.

- Gráfica comparando el tiempo medio para para InsertSort y BubbleSort, comentarios a la gráfica.



En este caso, el argumento es igual al de OBs medio, ya que simplemente, muestra resultado que hemos analizado en las clases de teoría, donde ambos algoritmos son cuadráticos y InsertSort funciona más rápido que BubbleSort en el caso medio.

- Gráfica comparando el tiempo peor para para InsertSort y BubbleSort, comentarios a la gráfica.



Está gráfica ilustra que, en el caso peor, el InsertSort es considerablemente más eficiente que BubbleSort.

6. Respuesta a las preguntas teóricas.

6.1 Pregunta 1

Como nos pide diseñar un programa que devuelve un número aleatorio equiprobable entre inf y sup, puesto que al hacer con la operación módulo no es posible que sea equiprobable, ya que si RAND_MAX es 4, y hacemos módulo 2: 0%2 ->0 , 1%2->1, 2%2->0, 3%2->1, 4%2->0, la probabilidad de que salga 0 es 60% y el 1 es 40%.

Por lo que aprovechamos otra forma de hacerlo, sin la operación módulo, como que **rand()** nos devuelve de forma equiprobable un número entre **0** y **RAND_MAX,** pues **rand()/(RAND_MAX+1)** nos devuelve un número equiprobable entre **[0, 1)**. Multiplicando por **(sup-inf+1)** nos devuelve un número entre **[0,sup-inf+1)** = **[0,sup-inf]**, así sumando inf, tenemos un número equiprobable perteneciente a [**inf,sup-inf+inf]=[inf,sup]**, que es lo que se pide.

Otro método de generar número aleatorio es que la función devuelva, inf+rand() %(sup-inf+1), pero la desventaja es lo que hemos mencionado anteriormente, **NO GENERA NÚMERO EQUIPROBABLES.**

6.2 Pregunta 2

El algoritmo de InsertSort ordenar bien porque en cada paso, asegura que la parte del arreglo que ya ha procesado esta ordenada. Al principio, el primer elemento de arreglo está solo, en consecuencia, ya está ordenada. Después, en cada iteración se toma el siguiente elemento y lo inserta en un lugar correcto dentro de arreglo que está ordenada y moviendo los elementos mayores hacia derecha en comparación con el nuevo elemento de tan forma el arreglo sigue quedando ordenada. De esta forma, en cada iteración se va introduciendo nuevos elementos para ordenar. Finalmente, cuando termina la interación todo el arreglo ha pasado por este proceso, en consecuencia, el resultado final esta ordenada.

6.3 Pregunta 3

El bucle exterior de InsertSort no actúa sobre el ultimo elemento de la tabla porque cuando el algoritmo llega a ultimo elemento, ya está ordenada automáticamente y no hace falta recorrer una iteración adicional para comprobar que está ordenada.

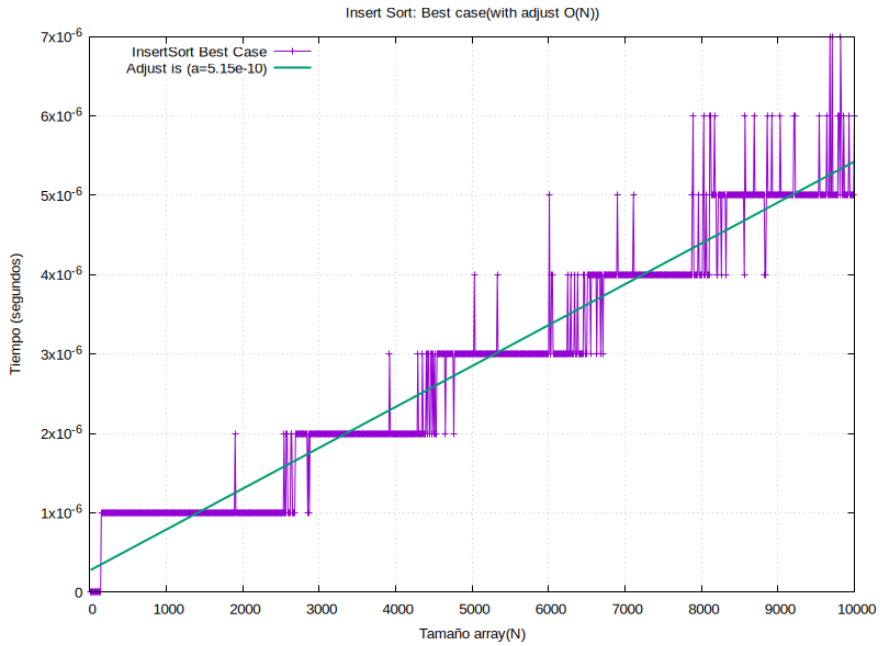
6.4 Pregunta 4

La operación básica de InsertSort es la comparación de entre los elementos, porque para saber dónde insertamos el elemento, el algoritmo comparar el elemento nuevo con los que ya están ordenados y este proceso se repite en cada intención varias veces. Según la definición de operación básica, la comparación de InsertSort lo cumple esa definición.

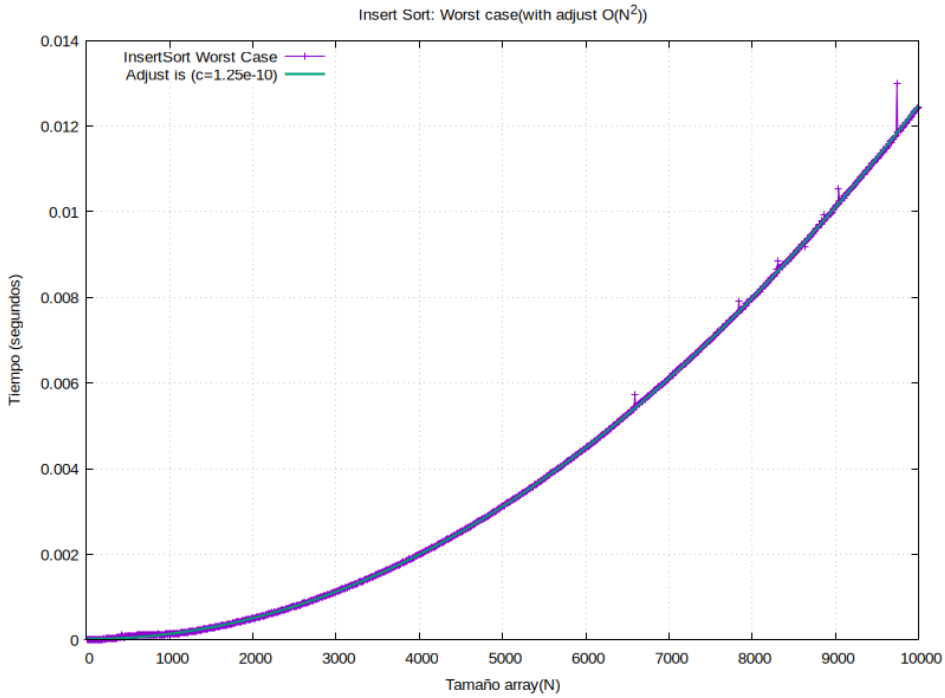
6.5 Pregunta 5

- InsertSort:

  En el caso mejor, hacemos una gráfica de tiempo-tamaño_array y usamos la Gnuplot para ajustarlo (en este caso una recta). Donde vemos que sí que es O(N) en el caso mejor.

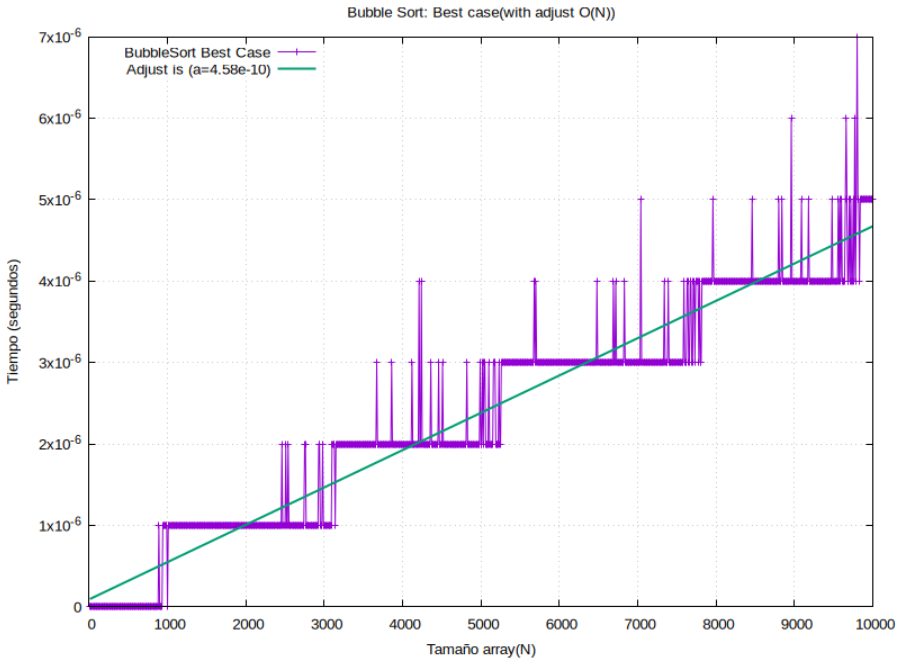


  En el caso peor, ajustamos la gráfica tiempo-tamaño_array con una función O(N^2). Así verificando que es de O(N^2) en el caso peor.

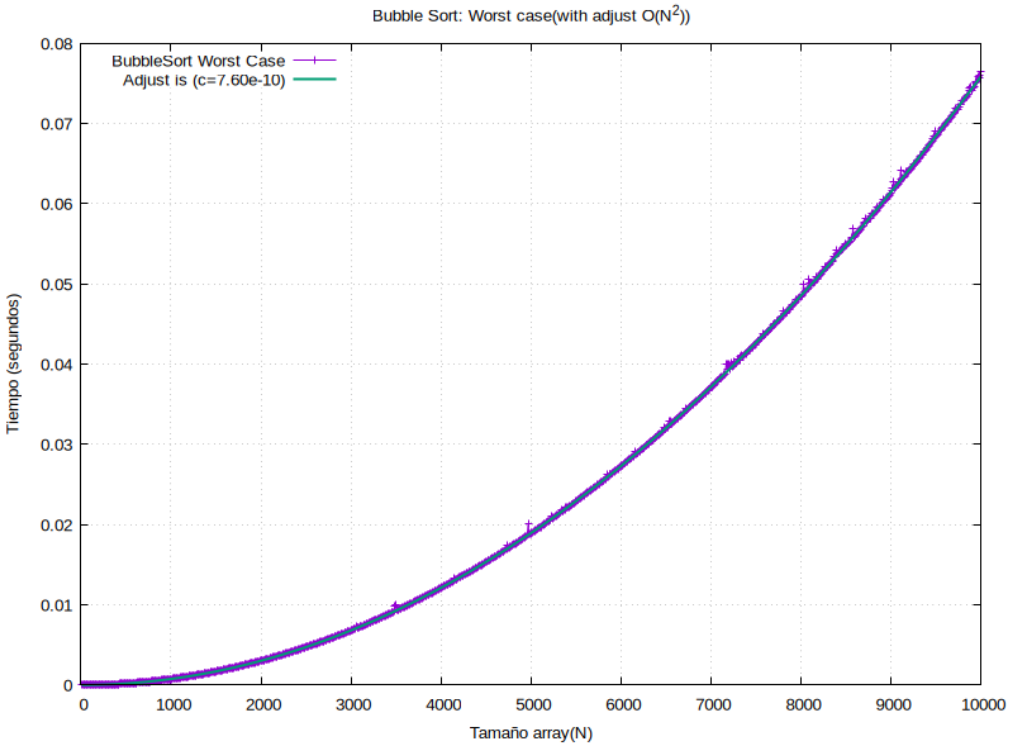


- BubbleSort:

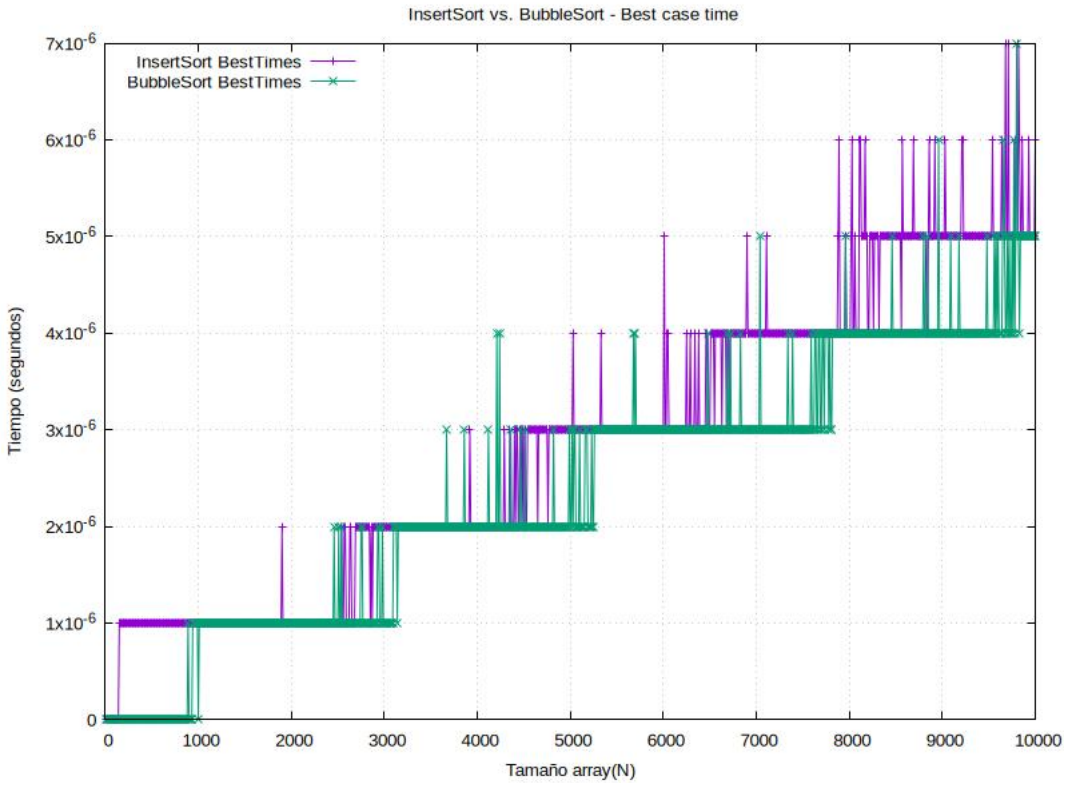
  En el caso mejor, al igual que el apartado anterior, hacemos una gráfica de tiempo-tamaño_array y usamos la Gnuplot para ajustarlo (en este caso una recta). Donde vemos que sí que es O(N) en el caso mejor.

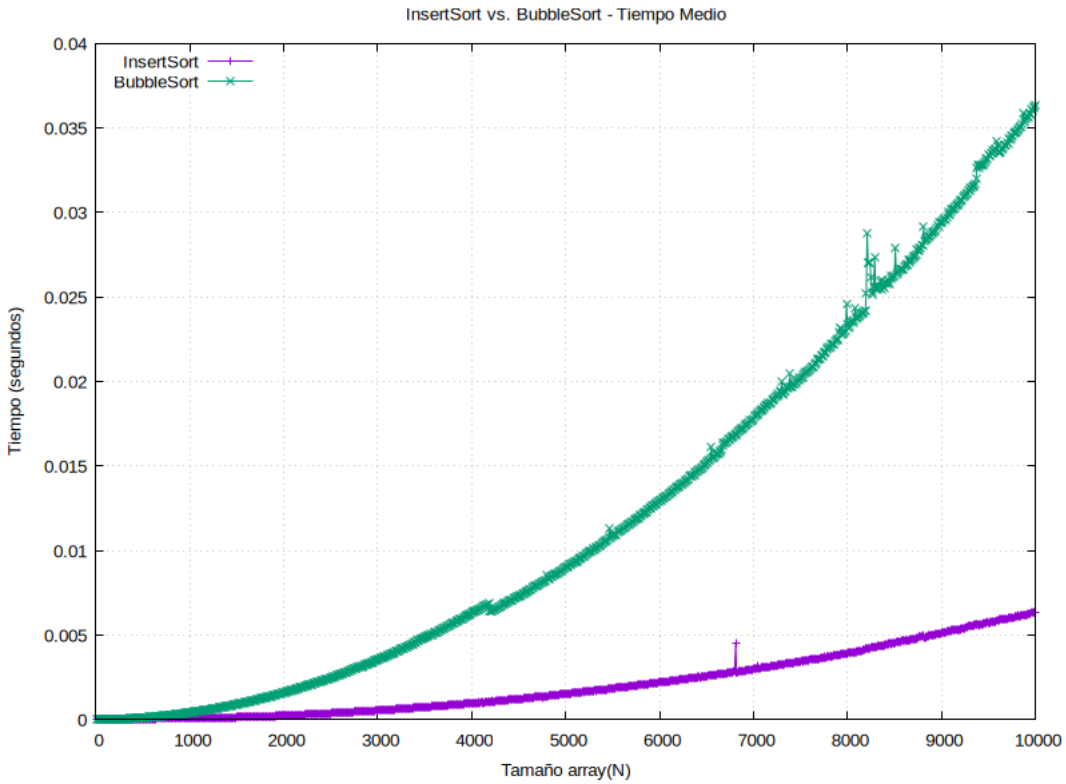


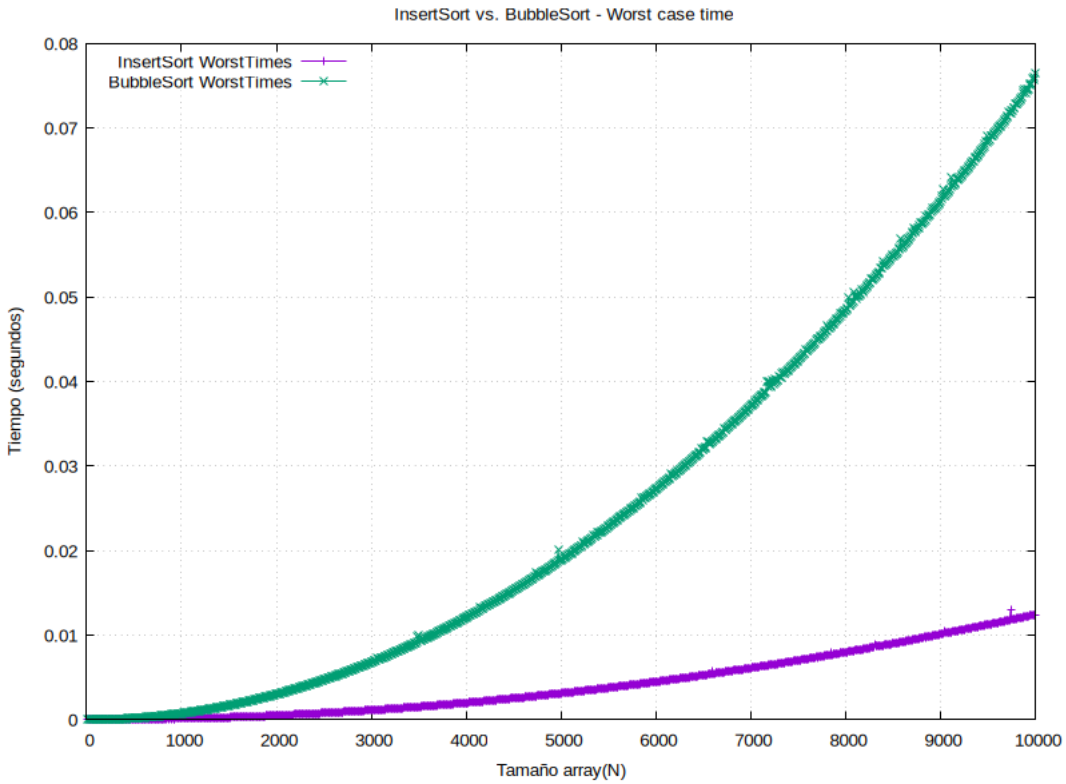
  En el caso peor, ajustamos la gráfica tiempo-tamaño_array con una función O(N^2). Así verificando que es de O(N^2) en el caso peor para BubbleSort.



6.5 Pregunta 6







- **Similitudes:** En el caso medio y peor, ambos algoritmos de ordenación tienen un comportamiento de O(N^2). Y en el mejor caso, ambos algoritmos son de comportamiento O(N). Y esto es debido a que ambos algoritmos son de comparación local, lo cual es imposible que tenga un rendimiento mejor en los tres casos. Y en los 3 casos, InsertSort es más eficiente que BubbleSort.

7. Conclusiones finales.

Esta práctica ha servido para contrastar de forma empírica el rendimiento teórico y práctico de dos algoritmos de ordenación cuadráticos: InsertSort y BubbleSort.

El objetivo principal era doble: implementar los algoritmos y comparar su eficiencia midiendo tanto el número de operaciones básicas (OBs) como el tiempo de ejecución real. Las conclusiones extraídas de los resultados obtenidos son claras:

De esta manera, conseguimos probar que InsertSort tiene un rendimiento notablemente mejor que BubbleSort. Así, hemos comprobado las conclusiones que hemos visto en la clase de teoría.

Sin embargo, encontramos también con problemas como medición de tiempo, ya que este puede no ser preciso a la hora de medir intervalos pequeños.