# Análisis de Algoritmos 2025/2026

# Práctica 1

**Rodrigo Díaz-Regañón Ureña**

**Daniel Martínez Fernández**

| Código | Gráficas | Memoria | Total |
|---|---|---|---|
| | | | |

# Índice:

# 1. Introducción.

En esta práctica se pretende estudiar el comportamiento de los algoritmos, desde la generación de datos de entrada hasta su eficiencia temporal. El objetivo principal es evaluar experimentalmente el rendimiento de distintos métodos de ordenación, comenzando con InsertSort y posteriormente comparándolo con BubbleSort. Además, se han implementado funciones en C que permiten generar permutaciones aleatorias, ordenarlas mediante distintos algoritmos, medir el tiempo medio de ejecución y el número de operaciones básicas realizadas.

# 2. Objetivos

### 2.1 Apartado 1

El objetivo de este apartado es comprender cómo funcionan algunos métodos de generación de números aleatorios e implementar uno propio. Así como hacer un estudio probabilístico de la aleatoriedad de este algoritmo, mediante un histograma.

### 2.2 Apartado 2

En este apartado se nos pide implementar una función para generar una permutación con el pseudocódigo dado. La generación de las permutaciones se basa en el algoritmo implementado para la generación de números aleatorios.

### 2.3 Apartado 3

En este apartado se nos pide realizar una función que utilizando la anterior implementación de permutaciones genere varias permutaciones equiprobables.

### 2.4 Apartado 4

En este apartado nos pide realizar la implementación del algoritmo InsertSort, siguiendo unas pautas marcadas, sabiendo reconocer cuál es su operación básica. El objetivo será entender el funcionamiento de este algoritmo

### 2.5 Apartado 5

En este apartado se nos pide implementar tres funciones, la primera de ellas guarda en la estructura time_aa el tamaño de la permutación, el número de permutaciones generadas, el tiempo medio de ordenación de las permutaciones, el promedio de operaciones básicas realizadas, el mínimo de operaciones básicas realizadas para la ordenación de una permutación y el máximo de operaciones básicas realizadas para la ordenación de una permutación.

Otra función a implementar es una en la que se imprimirá en un archivo toda la estructura time_aa para varios conjuntos de permutaciones cuyo orden va aumentando según el incremento introducido y parte de un orden dado hasta llegar a otro máximo. Para imprimirlos, se implementará otra función que realice una tabla donde se recojan todos los datos de la estructura time_aa, para cada elemento de un array de esa estructura.

### 2.6 Apartado 6

En este apartado se nos pide realizar la implementación del algoritmo de ordenación BubblueSort, siguiendo unas premisas dadas. El objetivo será entender el funcionamiento de este algoritmo, así como reconocer su OB.

## 3. Herramientas y metodología

### 3.1 Apartado 1

Para la realización de este apartado se implementaron las funciones a través de un portátil con Linux (Ubuntu)  y otro MacOS, utilizando Visual Studio como entorno de programación.

Todas las implementaciones se sometieron a un test, haciendo varias pruebas (cambiando el prompt para el comando exercise1_test del makefile) y llevándolo a situaciones extremas, es decir, limInf=limSup, limInf>limSup, etc.

Mediante la herramienta Valgrind se realizó un análisis completo para descartar cualquier tipo de problema de gestión de memoria, accesos inválidos…

### 3.2 Apartado 2

Para la realización de este apartado se implementaron las funciones a través de un portátil con Linux (Ubuntu)  y otro MacOS, utilizando Visual Studio como entorno de programación.

Todas las implementaciones se sometieron a un test, haciendo varias pruebas (cambiando el prompt para el comando exercise2_test del makefile) y llevándolo a situaciones extremas, es decir, size=0, size<0, etc.

Mediante la herramienta Valgrind se realizó un análisis completo para descartar cualquier tipo de problema de gestión de memoria, accesos inválidos…

Se ha detectado un problema, al introducir un numP<0. Al hacerlo se ejecutan muchísimas permutaciones, esto sucede porque en el archivo exercise2.c num se declara como un unsigned int, por lo que al guardar un número negativo en complemento a dos, el MSB es un 1, por lo que se harán 2 elevado al número de bits que tenga un unsigned int permutaciones como mínimo

### 3.3 Apartado 3

Para la realización de este apartado se implementaron las funciones a través de un portátil con Linux (Ubuntu)  y otro MacOS, utilizando Visual Studio como entorno de programación.

Todas las implementaciones se sometieron a un test, haciendo varias pruebas (cambiando el prompt para el comando exercise3_test del makefile) y llevándolo a situaciones extremas, es decir, size=0, size<0, etc.

Mediante la herramienta Valgrind se realizó un análisis completo para descartar cualquier tipo de problema de gestión de memoria, accesos inválidos…

Aquí también tenemos el problema cuando numP<0.

### 3.4 Apartado 4

Para la realización de este apartado se implementó la función Insertsort a través de un portátil con Linux (Ubuntu)  y otro MacOS, utilizando Visual Studio como entorno de programación.

La implementación de este algoritmo se sometió a un test, haciendo varias pruebas (cambiando el prompt para el comando exercis4_test del makefile) y llevándolo a situaciones extremas, es decir, size=0, size<0, etc.

Mediante la herramienta Valgrind se realizó un análisis completo para descartar cualquier tipo de problema de gestión de memoria, accesos inválidos…

### 3.5 Apartado 5

Para la realización de este apartado se implementaron las funciones pedidas sobre time, a través de un portátil con Linux (Ubuntu)  y otro MacOS, utilizando Visual Studio como entorno de programación.

La implementación de estas funciones se sometieron a un test, usando el algoritmo de ordenación previamente implementado, Insertsort, haciendo varias pruebas (cambiando el prompt para el comando exercise5_test del makefile) y llevándolo a situaciones extremas, es decir, num_min<0 ó num_max<0, num_min>=num_max, incr<=0, numP<=0, outputFile un fichero sin terminación, etc.

Mediante la herramienta Valgrind se realizó un análisis completo para descartar cualquier tipo de problema de gestión de memoria, accesos inválidos…

### 3.6 Apartado 6

Para la realización de este apartado se implementó la función BubbleSort a través de un portátil con Linux (Ubuntu)  y otro MacOS, utilizando Visual Studio como entorno de programación.

La implementación de este algoritmo se sometió a un test, haciendo varias pruebas (cambiando el prompt para el comando exercise4_test del makefile) y llevándolo a situaciones extremas, es decir, size=0, size<0, etc.

Además se usó exercise5_test, para probar la funciones de time con este algoritmo, llevándolo a situaciones extremas, num_min<0 ó num_max<0, num_min>=num_max, incr<=0, numP<=0, outputFile un fichero sin terminación, etc.

Mediante la herramienta Valgrind se realizó un análisis completo para descartar cualquier tipo de problema de gestión de memoria, accesos inválidos…

# 4. Código fuente

## 4.1 Apartado 1

```c
#include "permutations.h"
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
/*CONSTANTES PARA LA FUNCION QUE GENERA RANDOM NUMBERS*/
#define IA 16807 /*Multiplicador de Park and Miller*/
#define IM 2147483647/* Primo 2^31 - 1 muy grande que será nuestro módulo y nos dará una secuencia de 2^31-2 números*/
#define AM (1.0/IM)/* Convierte el número generado a un flotante en [0,1)*/
#define IQ 127773 /* IM/IA lo utilizaremos para evitar overflow */
#define IR 2836 /* IM%IA utilizado para evitar overflow*/
#define NTAB 32 /* Tamaño de la tabla de shuffle Bays-Durham*/
#define NDIV (1+(IM-1)/NTAB)/* Factor para indexar la tabla de shuffle*/
#define EPS 1.2e-7 /* epsilon lo suficientemente pequeño*/
#define RNMX (1.0-EPS)/* evita que el generador devuelva exactamente 1*/

float ran1(long *idum){
  int j;
  long k;
  static long iy=0;
  static long iv[NTAB];
  float temp;

  if (*idum <= 0 || !iy)
  {
    if (-(*idum) < 1) *idum = 1;
    else *idum = -(*idum);

    for (j = NTAB+7; j >= 0; j--) {
        k = (*idum) / IQ;
        *idum = IA * (*idum - k * IQ) - IR * k;
        if (*idum < 0) *idum += IM;
        if (j < NTAB) iv[j] = *idum;
    }
    iy = iv[0];
  }
    /*Generación del numero aleatorio*/
    k = (*idum) / IQ;
    *idum = IA * (*idum - k * IQ) - IR * k;
    if (*idum < 0) *idum += IM;
    j = iy / NDIV;
    iy = iv[j];
    iv[j] = *idum;

    if ((temp = AM * iy) > RNMX) return RNMX; else return temp;
}


int random_num(int inf, int sup)
{
  static long seed = 0;
  int rango;
  float rand;

  if(inf>sup){
    fprintf(stderr, "El índice inferior no puede ser mayor que el superior.");
    return ERR;
  }
  if(seed == 0){
    seed = -(long)time(NULL); /*Va a ser la semilla inicial*/
  }
  rango = sup - inf + 1;
  rand = ran1(&seed);
  return inf + (int)(rand*rango);
}
```

### 4.2 Apartado 2

```c
int* generate_perm(int N)
{
  int i, *perm,e_aux,i_aux;
  if(!(perm =(int*)calloc(N, sizeof(int)))){
    return NULL;
  }
  for(i=1;i<=N;i++){
    perm[i-1] = i;
  }
  for(i=0;i<N;i++){
    e_aux = perm[i];
    i_aux = random_num(0,N-1);
    perm[i] = perm[i_aux];
    perm[i_aux] = e_aux;
  }

  return perm;
}
```

### 4.3 Apartado 3

```c
int** generate_permutations(int n_perms, int N)
{
  int **array_perm = NULL, i, j;

  if (n_perms <=0 || N<= 0)
  {
    return NULL;
  }

  array_perm = (int**)calloc(n_perms,sizeof(int*))
  if (array_perm == NULL)
  {
    return NULL;
  }

  for (i=0; i<n_perms; i++)
  {
    array_perm[i]= generate_perm(N);

    if (array_perm[i]==NULL)
    {
      for (j = 0; j < i; j++)
      {
        free(array_perm[j]);
      }
    free(array_perm);
    return NULL;
    }
  }
  return array_perm;
}
```

### 4.4 Apartado 4

```c
int InsertSort(int *array, int ip, int iu)
{
  int i, j, OB = 0, aux;

  if (array == NULL)
  {
    fprintf(stderr, "El array es invalido.\n");
    return ERR;
  }
  if (ip < 0 || iu < 0)
  {
    fprintf(stderr, "Los indices tienen que ser positivos.");
    return ERR;
  }
  if (ip > iu)
  {
    fprintf(stderr, "El primer indice tiene que ser mayor que el ultimo.\n");
    return ERR;
  }


  for (i = ip + 1; i <= iu; i++)
  {
    aux = array[i];
    j = i - 1;
    while (j >= ip && array[j] > aux)
    {
      array[j + 1] = array[j];
      OB++;
      j--;
    }
    if (j >= ip) OB++;
    array[j + 1] = aux;
  }

  return OB;
}
```

### 4.5 Apartado 5

```c
short average_sorting_time(pfunc_sort metodo,
                           int n_perms,
                           int N,
                           PTIME_AA ptime)
{
  int i, max_OB = -1, min_OB = -1, OB_aux, **array_perms;
  double sum_OB = 0;
  clock_t start, end;

  if (metodo == NULL || ptime == NULL || n_perms <= 0 || N <= 0)
  {
    fprintf(stderr, "Las variables introducidas no son válidas.");
    return ERR;
  }

  ptime->n_elems = n_perms;
  ptime->N = N;

  array_perms = generate_permutations(n_perms, N);
  if (array_perms == NULL)
  {
    return ERR;
  }
  start = clock();
  for (i = 0; i < n_perms; i++)
  {

    OB_aux = metodo(array_perms[i], 0, N - 1);

    if (max_OB == -1)
    {
      max_OB = OB_aux;
    }
    if (min_OB == -1)
    {
      min_OB = OB_aux;
    }

    if (OB_aux > max_OB)
    {
      max_OB = OB_aux;
    }
    if (OB_aux < min_OB)
    {
      min_OB = OB_aux;
    }
    sum_OB += OB_aux;

    free(array_perms[i]);
  }
  end = clock();
  ptime->average_ob = sum_OB / n_perms;
  ptime->max_ob = max_OB;
  ptime->min_ob = min_OB;
  ptime->time = (double)(end - start) /CLOCKS_PER_SEC;
  free(array_perms);
  return OK;
}
```

```c
short generate_sorting_times(pfunc_sort method, char *file,
                             int num_min, int num_max,
                             int incr, int n_perms)
{
  int i, n_times;
  PTIME_AA ptime = NULL;

  if (!method || num_min <= 0 || num_max < num_min || incr <= 0)
  {
    fprintf(stderr, "Las variables introducidas no son válidas.\n");
    return ERR;
  }

  n_times = ((num_max - num_min) / incr) + 1;

  if (!(ptime = (PTIME_AA)calloc(n_times, sizeof(TIME_AA))))
  {
    fprintf(stderr, "Error al reservar memoria para la estructura del tiempo.\n");
    return ERR;
  }
  for (i = 0; i < n_times; i++)
  {
    int N = num_min + i * incr;
    if (average_sorting_time(method, n_perms, N, &ptime[i]) == ERR)
    {
      free(ptime);
      return ERR;
    }
  }
  if (save_time_table(file, ptime, n_times) == ERR)
  {
    free(ptime);
    return ERR;
  }

  free(ptime);
  return OK;
}
```

```c
short save_time_table(char *file, PTIME_AA ptime, int n_times)
{
  FILE *fout = NULL;
  int i;

  if (!ptime || n_times <= 0 || !file)
  {
    fprintf(stderr, "Las variables introducidas no son válidas.");
    return ERR;
  }

  if (!(fout = fopen(file, "w")))
  {
    fprintf(stderr, "Error al abrir el fichero");
    return ERR;
  }

  fprintf(fout, "%-6s %-12s %-14s %-8s %-8s\n", "N", "Time", "Average_OB", "Max_OB", "Min_OB");

  for (i = 0; i < n_times; i++)
  {
    fprintf(fout, "%-6i %-12.6f %-14.2f %-8i %-8i\n",
            ptime[i].N,
            ptime[i].time,
            ptime[i].average_ob,
            ptime[i].max_ob,
            ptime[i].min_ob);
  }

  fclose(fout);
  return OK;
}
```

### 4.6 Apartado 6

```c
int BubbleSort(int *array, int ip, int iu)
{
  int flag = 1;
  int i = iu;
  int j = 0;
  int aux = 0;
  int OB = 0;

  if (array == NULL)
  {
    fprintf(stderr, "El array es invalido.\n");
    return ERR;
  }
  if (ip < 0 || iu < 0)
  {
    fprintf(stderr, "Los indices tienen que ser positivos.");
    return ERR;
  }
  if (ip > iu)
  {
    fprintf(stderr, "El primer indice tiene que ser mayor que el ultimo.\n");
    return ERR;
  }

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

# 5. Resultados, Gráficas

## 5.1 Apartado 1

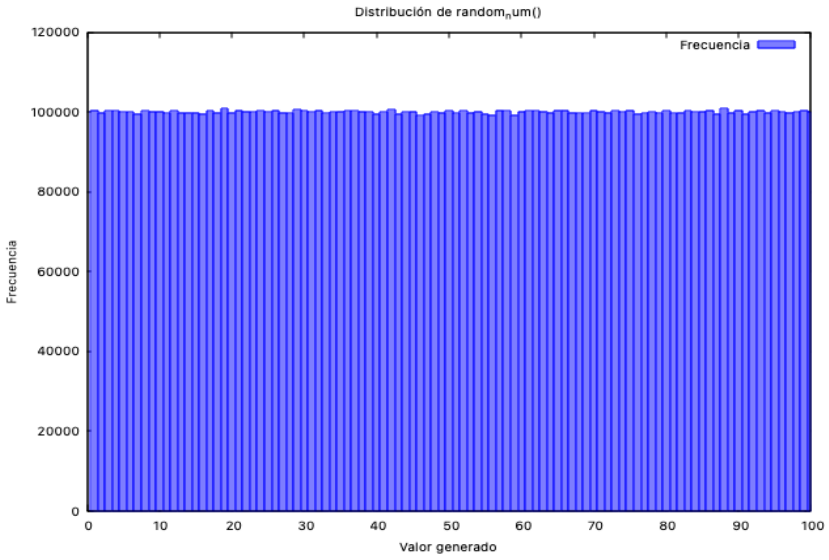
Sobre la función de números aleatorios, hemos realizado una prueba en la que se han generado 10.000.000 números aleatorios entre 1 y 100. La desviación típica obtenida fue la siguien:

$$\sigma_{\text{experimental}} = 28.8718$$

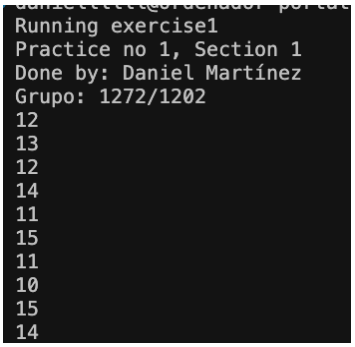
Por otro lado la teórica es la siguiente: $$\sigma_{\text{teórica}} = \sqrt{\frac{(b-a+1)^2 - 1}{12}}$$

$$\sigma_{\text{teórica}} = \sqrt{\frac{(100)^2 - 1}{12}} = \sqrt{\frac{9999}{12}} = 28.866$$

Si calculamos por tanto el error nos sale que es aproximadamente del 0.02%. Por tanto, nuestro algoritmo de números aleatorios produce una distribución uniforme y equiprobable. La gráfica obtenida fue la siguiente:



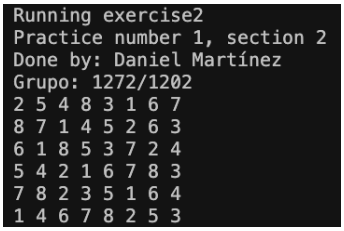
Cuando hacemos el exercise1_test, donde se generan numN números aleatorios en el rango (limInf, limSup), con los parámetros -limInf 10 -limSup 15 -numN 10.




Obtenemos la siguiente salida:

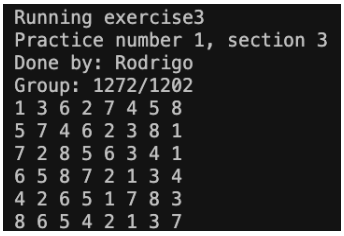
### 5.2 Apartado 2

El resultado obtenido tras realizar el test exercise2, donde se realizan numP permutaciones de tamaño size, con estos parámetros, -size 8 -numP 6, es el siguiente:



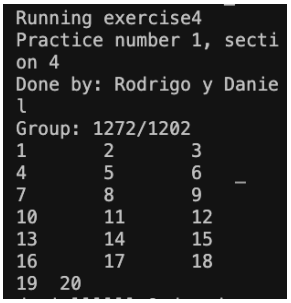

### 5.3 Apartado 3

El resultado obtenido tras realizar el test exercise3, donde se realizan numP permutaciones de tamaño size (es decir lo mismo que en el anterior test, pero usando funciones distintas, lógicamente) con los mismos parámetros, -size 8 -numP 6, es el siguiente:




### 5.4 Apartado 4

El resultado obtenido tras realizar el test exercise4, donde se realiza la ordenación de una permutación de tamaño size, los parámetros introducidos han sido -size 20 y la salida fue la siguiente:




### 5.5 Apartado 5

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

Para este apartado se han usado los siguientes datos para sacar los datos experimentales de InsertSort: tamaño mínimo de la permutación (1), tamaño máximo de la permutación (20), incremento de tamaño entre cada permutación (1) y número de permutaciones (100000).

Como hemos visto en teoría los tiempos para Insert sort, son los siguientes:

$$W_{IS}(N) = \frac{N(N-1)}{2} \qquad \mathrm{BIS}(N) = N - 1 \qquad \mathrm{AIS}(N) = \frac{N^2}{4} + \frac{3N}{4} - H_N$$

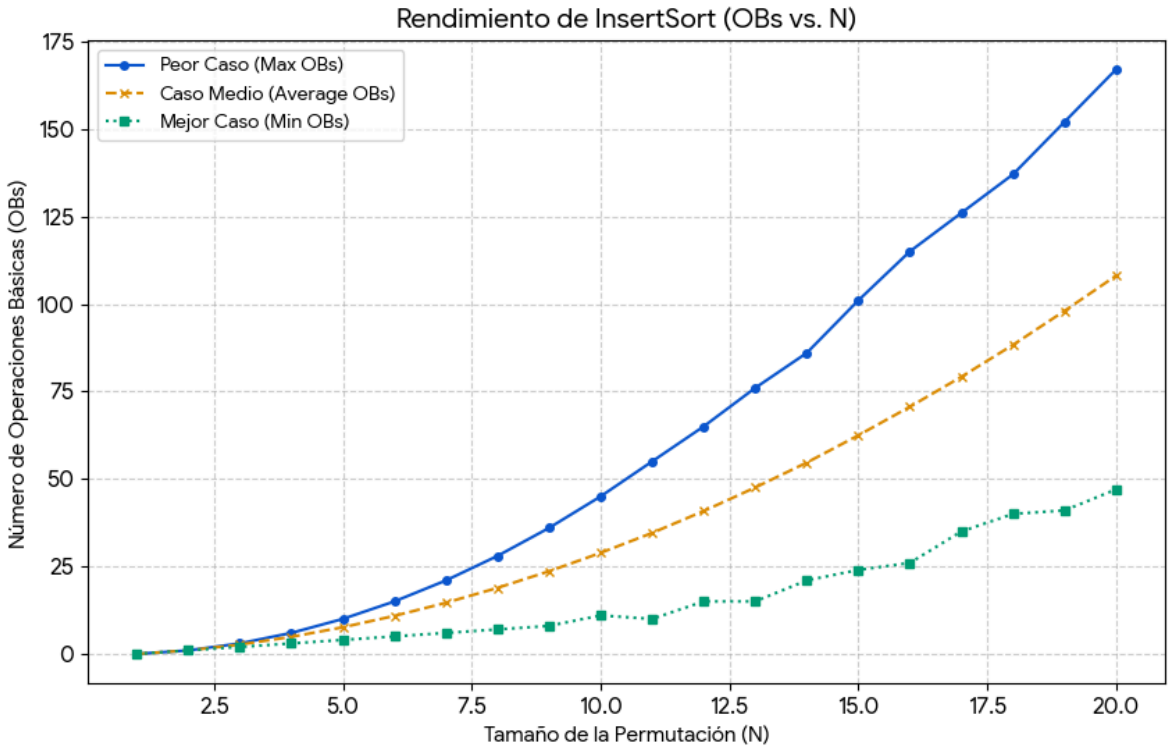
Por lo que podemos deducir que:

$$W_{is}(N) = N^2/2 + O(N)$$

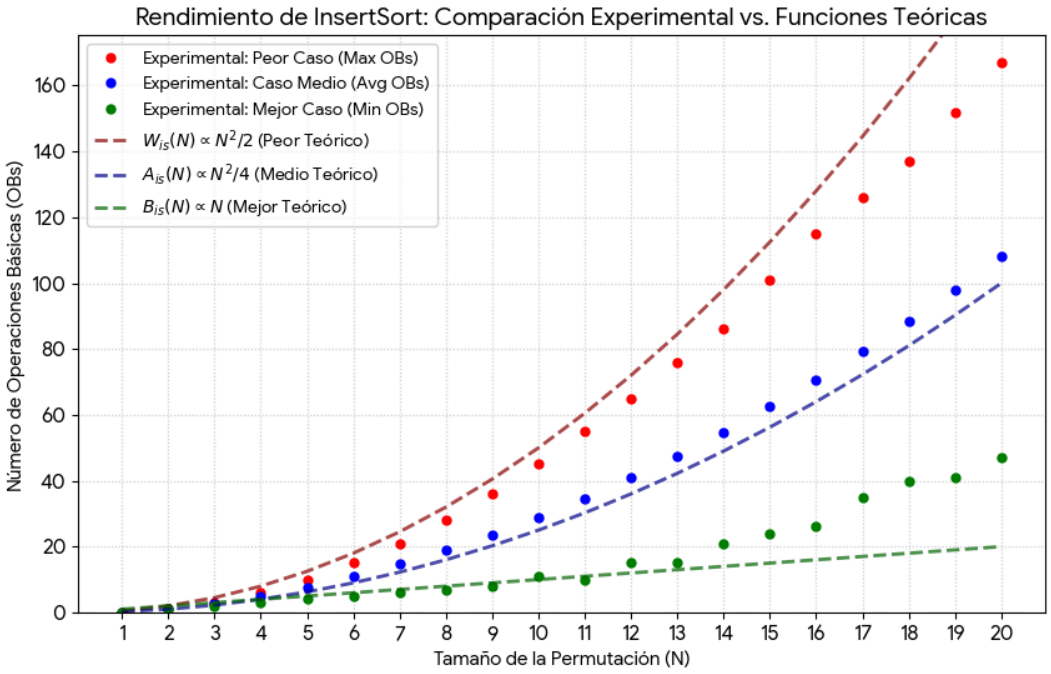
$$B_{is}(N) = O(N)$$

$$A_{is}(N) = N^2/4 + O(N)$$

Primero vamos a representar los datos experimentales:



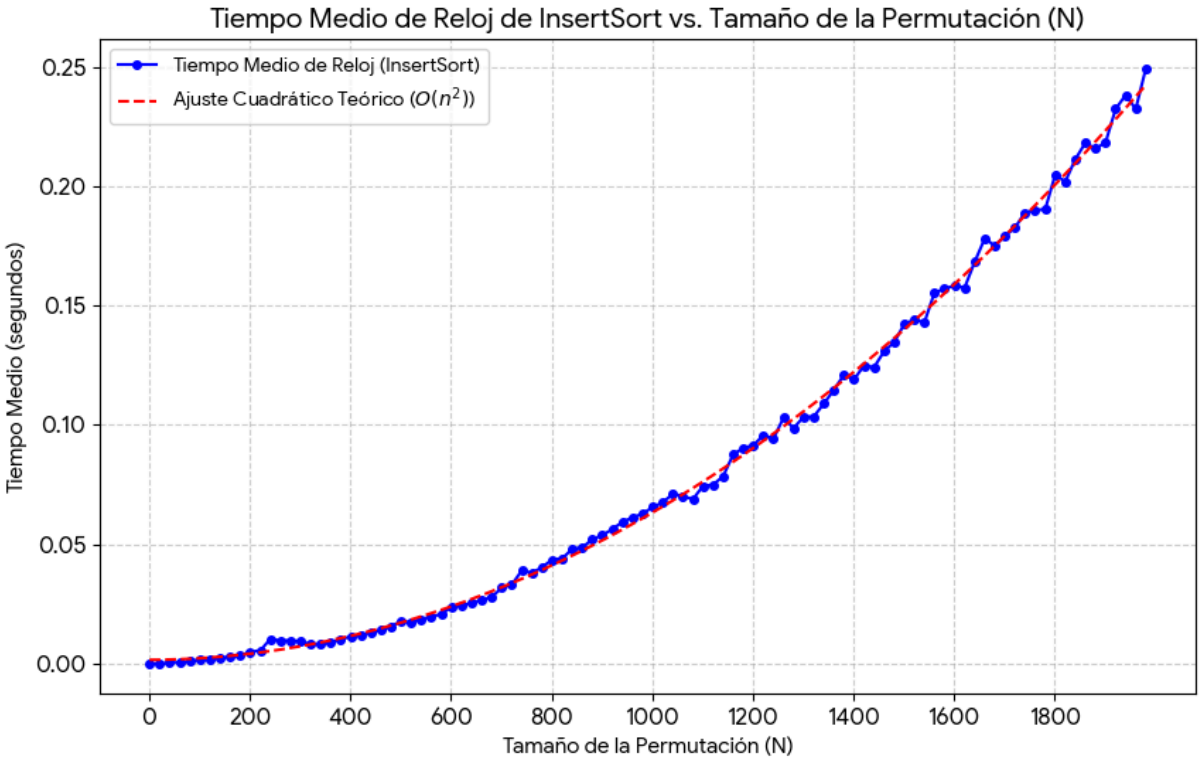

Para observar que los datos experimentales concuerdan con los datos teóricos, se ha creado otra gráfica que junto las funciones teóricas con los datos obtenidos en laboratorio:




Se puede observar que los datos experimentales se adaptan a los teóricos.

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

Para este apartado se han usado los siguientes datos para sacar los datos experimentales de InsertSort: tamaño mínimo de la permutación (1), tamaño máximo de la permutación (2000), incremento de tamaño entre cada permutación (20) y número de permutaciones (100).

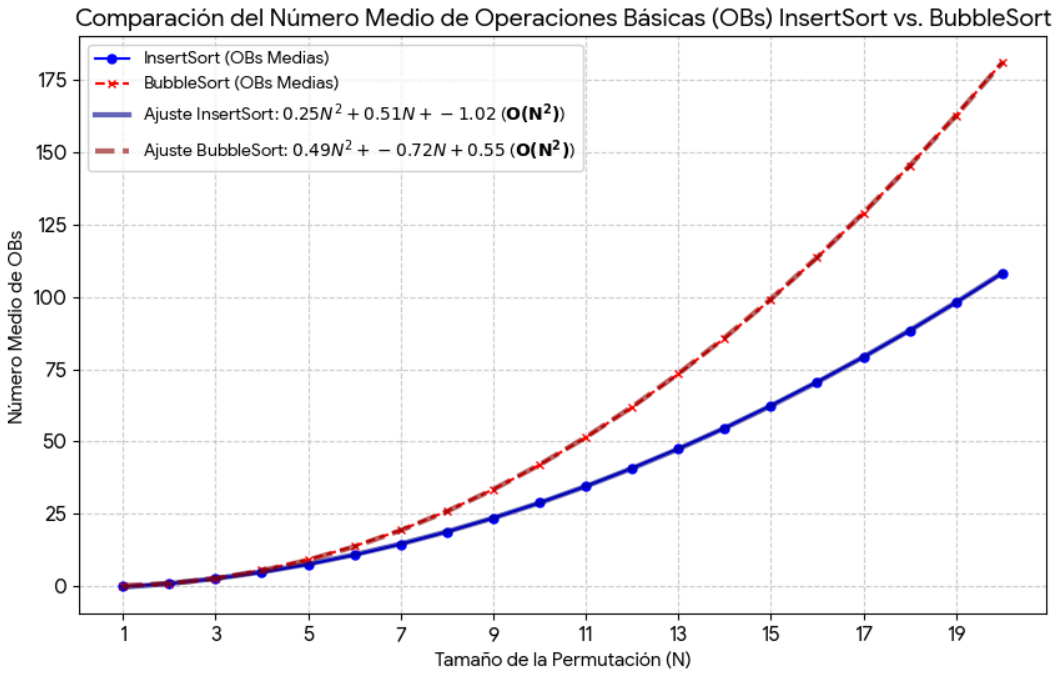



En esta gráfica observamos como la gráfica sigue una evidente función cuadrática, esta función cuadrática se ve en líneas discontinuas.

### 5.6. Apartado 6.

Para estos apartados se han usado los siguientes datos para sacar los datos experimentales de InsertSort: tamaño mínimo de la permutación (1), tamaño máximo de la permutación (20), incremento de tamaño entre cada permutación (1) y número de permutaciones (100000).

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

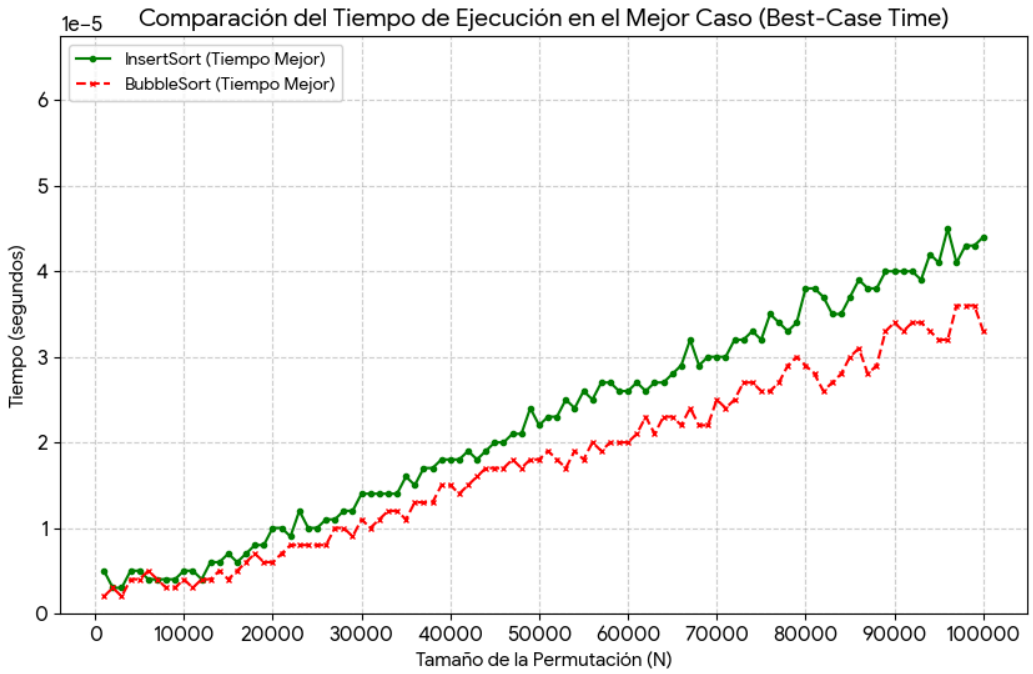



En esta gráfica se observa como InsertSort es algo más eficiente que el BubbleSort, Pues como hemos visto en teoría, $A_{BS} = N^2/2 + O(N)$ a diferencia de $A_{is}(N) = N^2/4 + O(N)$.

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

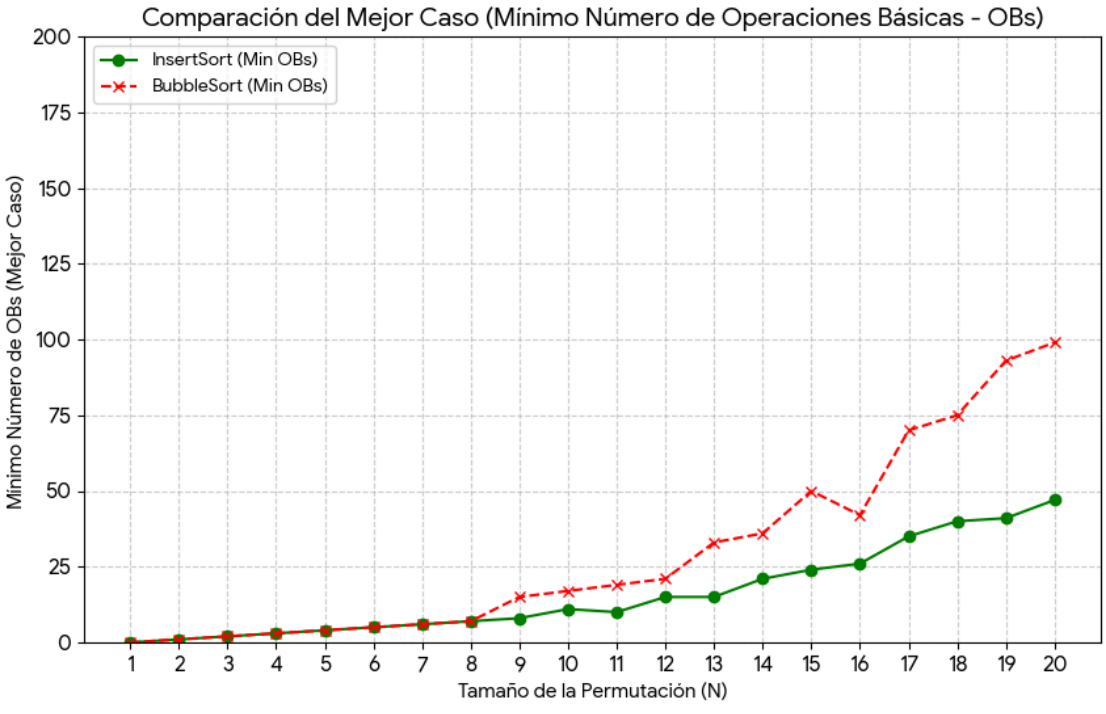
Dada la ambigüedad del enunciado, hemos decidido hacer dos gráficas, una en la que usemos los datos que nos da la tabla al hacer el ejercicio 5 (se mide el tiempo en OBs) lo cual tendrá un problema que explicaremos, y otra forma en la que forzamos el mejor caso para permutaciones de size entre 1.000 y 100.000 con incremento de 1.000, y medimos el tiempo en segundos que se tarda en completar el algoritmo.

- **Tiempo en segundos:**




Observamos como ambos algoritmos siguen un crecimiento lineal O(N) con picos, debido a diversos factores que no podemos controlar (como la velocidad de procesamiento del ordenador).

- **Tiempo en OBs usando exercise5:**



En esta gráfica el tiempo mejor debería ser lineal por lo visto en teoría, de hecho es así hasta el tamaño 9, esto se debe a que el tiempo mejor aquí está haciendo referencia a la permutación de n elementos que esté "más ordenada".

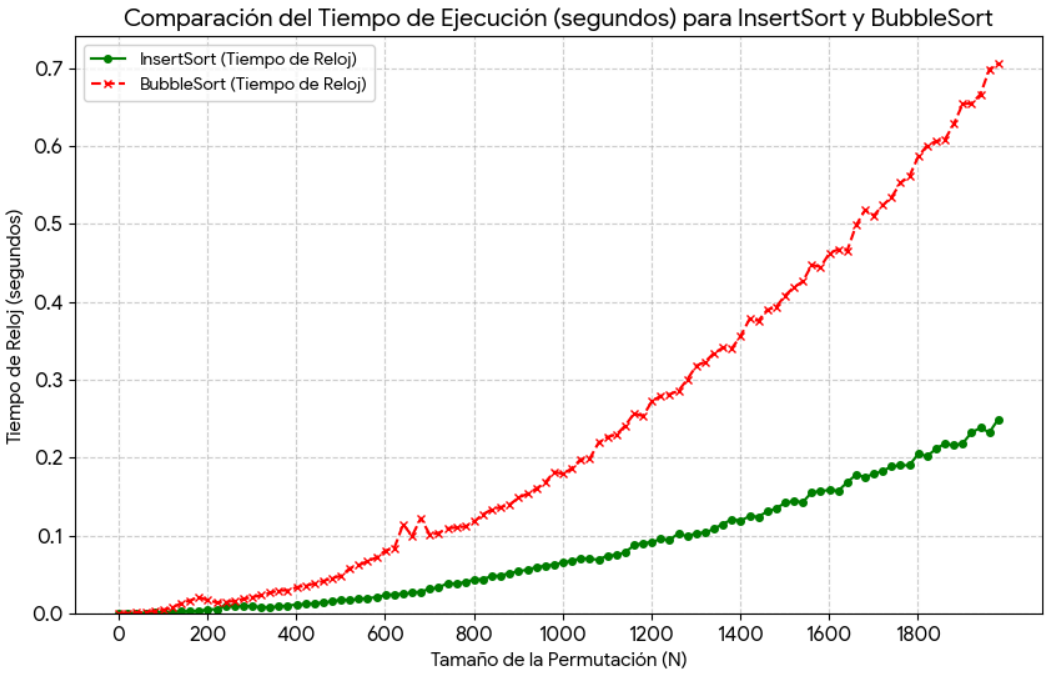
Para que la gráfica fuera lineal, el conjunto de permutaciones para cada tamaño n, debería de contener a la permutación (1,2,…,n), cuyo coste de ordenación es n (el menor). Sin embargo a partir de el tamaño 9 esto no se da, ya que el número de permutaciones dentro de ese $S_n$ es n!. Por tanto para n=9 hay 362.880 permutaciones, como el número de permutaciones hecho es de 100.000, la probabilidad de que se haya generado la permutación (1,2,3,4,5,6,7,8,9) no llega al 30%. Conforme n va creciendo, esta probabilidad es más pequeña y por tanto la gráfica dejará de ser lineal y adoptará un crecimiento mayor dependiendo de la eficiencia del algoritmo para casos más desordenados, por lo que se vuelve a corroborar que InsertSort es más óptimo.

- **Conclusión**

Fijándonos en la primera gráfica que es la que nos da la información para determinar qué algoritmo es más óptimo para el mejor caso, afirmamos que Bubblesort es más eficiente que Insertsort en el mejor caso, ya que la recta a la que se asemeja tiene una pendiente mayor a la del Insertsort.

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

Para este apartado, se han usado los siguientes datos para sacar los datos experimentales de BubbleSort e InsertSort: tamaño mínimo de la permutación (1), tamaño máximo de la permutación (1000), incremento de tamaño entre cada permutación (20) y número de permutaciones (1000).

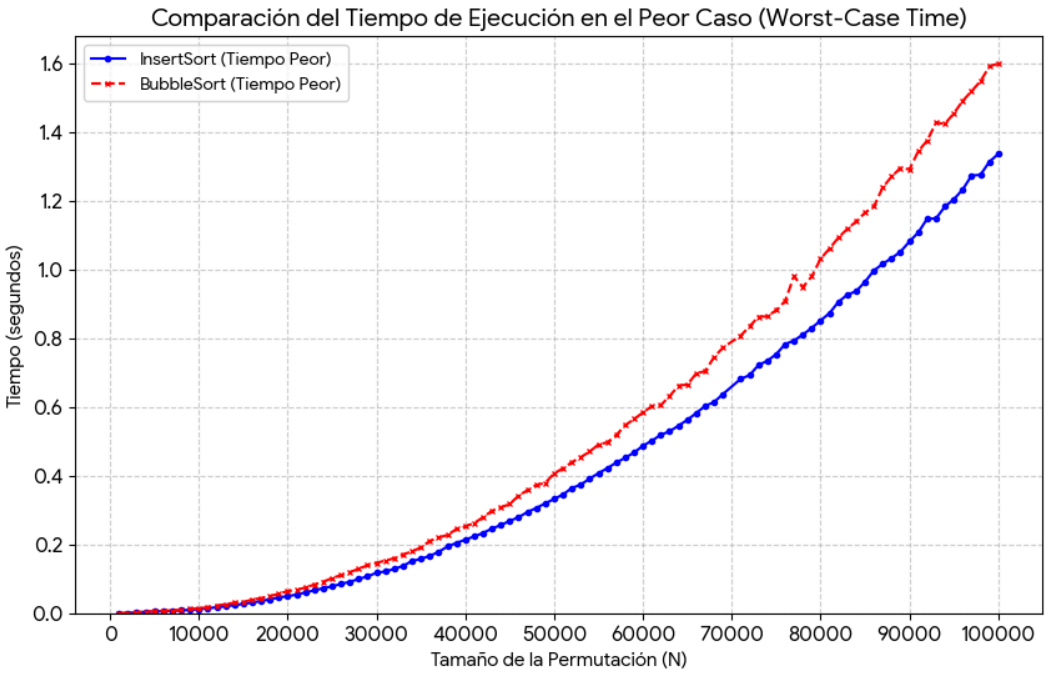



Esta gráfica como era de esperar se asemeja al tiempo medio en OBs. La diferencia es que aquí le estamos dando un valor temporal a las operaciones básicas. Por tanto la comparación es igual a la antes comentada.

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

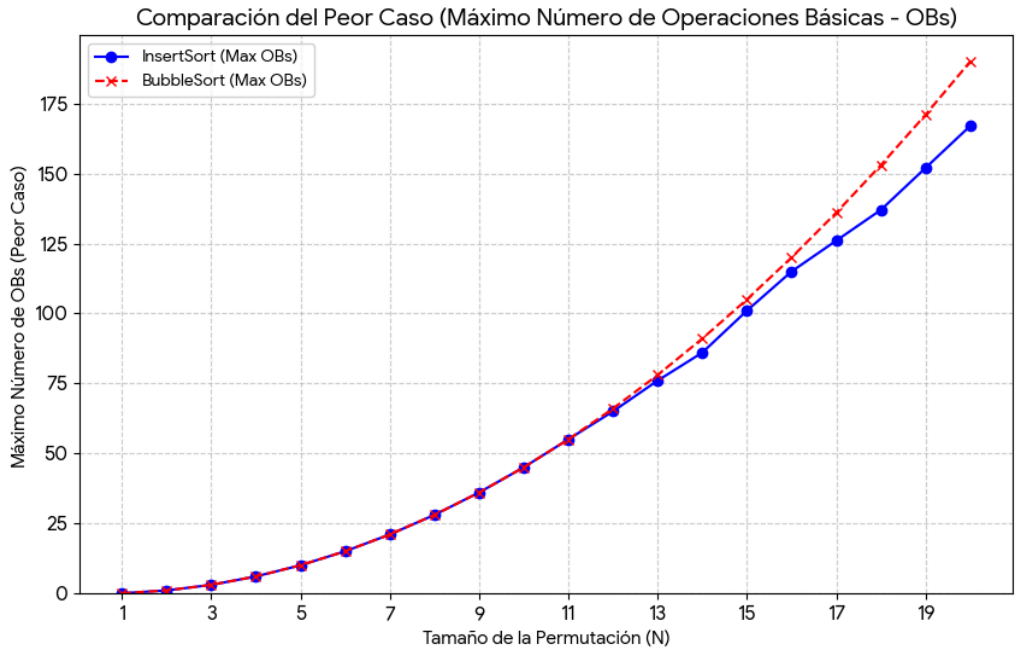
Dada la ambigüedad del enunciado, hemos decidido hacer dos gráficas, una en la que usemos los datos que nos da la tabla al hacer el ejercicio 5 (se mide el tiempo en OBs) lo cual tendrá un problema que explicaremos, y otra forma en la que forzamos el peor caso para permutaciones de size entre 1.000 y 100.000 con incremento de 1.000, y medimos el tiempo en segundos que se tarda en completar el algoritmo.

- **Tiempo en segundos:**




Observamos como ambos algoritmos siguen un crecimiento cuadrático $O(N^2)$ bastante regular, exceptuando de algunos puntos que pueden ser causados por la velocidad de procesamiento del ordenador. También se puede observar que InsertSort en el peor caso es mejor algoritmo que BubbleSort.

- **Tiempo en OBs:**



A diferencia del mejor caso O(N), la gráfica del peor caso exhibe el comportamiento que la teoría predice: un crecimiento marcadamente cuadrático $O(N^2)$ para ambos algoritmos. El peor caso ocurre cuando la permutación de entrada está ordenada inversamente, obligando al algoritmo a realizar el máximo número posible de operaciones. La curva ascendente y parabólica de ambas líneas confirma que el costo se escala con el cuadrado del tamaño de la permutación N.

En los valores iniciales de N (aproximadamente hasta N=13), las curvas de InsertSort y BubbleSort son casi idénticas y están superpuestas. Esto se debe a que la probabilidad de que la muestra de 100,000 permutaciones contenga la permutación inversamente ordenada (el peor caso) es similar para ambos en ese rango. La similitud inicial es la manifestación empírica de que ambos comparten la misma complejidad teórica $O(N^2)$ .

A partir de N=13, la curva de InsertSort (azul) comienza a separarse y se mantiene por debajo de la curva de BubbleSort (rojo). Esta divergencia es clave, ya que demuestra que, aunque la complejidad asintótica es la misma, la eficiencia real de InsertSort es superior. Su función cuadrática tiene un factor constante menor que la de BubbleSort. Esto se debe a que InsertSort, incluso en el peor caso, realiza operaciones de desplazamiento menos costosas en OBs totales que los intercambios y comparaciones repetitivas de BubbleSort, corroborando que InsertSort es más óptimo dentro de los algoritmos de orden O(N²).

- **Conclusión:**

Fijándonos en la primera gráfica que es la que nos da la información para determinar qué algoritmo es más óptimo para el peor caso, afirmamos que Insertsort es más eficiente que Bubblesort en el peor caso, ya que en todo momento se encuentra por debajo de los tiempos de Bubblesort.

# 6. Respuesta a las preguntas teóricas.

**6.1 Justifica tu implementación de aleat num ¿en qué ideas se basa? ¿de qué libro/artículo, si alguno, has tomado la idea? Propón un método alternativo de generación de números aleatorios y justifica sus ventajas/desventajas respecto a tu elección.**

El método elegido para crear números aleatorios ha sido el algoritmo de Park y Miller para crear números aleatorios, junto al shuffle/barajeo de Bays-Durham. Explicándolo rápidamente, este método se basa en, tras un calentamiento, llenar una tabla con "semillas" que serán números aleatorios creados con el algoritmo de Park y Miller, y posteriormente se seleccionará el último número generado y se convertirá en un índice dentro del rango de la tabla, este índice será el elemento seleccionado de la tabla para ser convertido a un float en [0,1) que será devuelto como nuestro número aleatorio.

Nótese que ese elemento será reemplazado en la tabla por otro generado a partir de este con el algoritmo de Park y Miller.

Las constantes usadas se explican en este pantallazo:

```
/*CONSTANTES PARA LA FUNCION QUE GENERA RANDOM NUMBERS*/
#define IA 16807 /*Multiplicador de Park and Miller*/
#define IM 2147483647/* Primo 2^31 - 1 muy grande que será nuestro módulo y nos dará una secuencia de 2^31-2 números*/
#define AM (1.0/IM)/* Convierte el número generado a un flotante en [0,1)*/
#define IQ 127773 /* IM/IA lo utilizaremos para evitar overflow */
#define IR 2836 /* IM%IA utilizado para evitar overflow*/
#define NTAB 32 /* Tamaño de la tabla de shuffle Bays-Durham*/
#define NDIV (1+(IM-1)/NTAB)/* Factor para indexar la tabla de shuffle*/
#define EPS 1.2e-7 /* epsilon lo suficientemente pequeño*/
#define RNMX (1.0-EPS)/* evita que el generador devuelva exactamente 1*/
```

En esta captura de pantalla, se muestra la explicación línea por línea de la función

```c
/**
 * @brief Función privada que nos dará el número random entre 0 y 1
 *
 * @param idum puntero a la seed
 * @return un decimal entr 0 y 1 sin incluir este ultimo
 */
float ran1(long *idum){
  int j; /*indice para la tabla shuffle*/
  long k; /* Variable auxiliar para el overflow*/
  static long iy=0; /*Numero previamente generado devuelto desde la tabla shuffle*/
  static long iv[NTAB];/* tabla shuffle*/
  float temp;/*Numero aleatorio que devolveremos*/

  if (*idum <= 0 || !iy)/* se inicializa si la semilla es negativa o si iy aún no se ha definido*/
  {
    if (-(*idum) < 1) *idum = 1;  /* Aseguramos que la semilla sea positiva y distinta de cero*/
    else *idum = -(*idum);

    for (j = NTAB+7; j >= 0; j--) { /* Se hace un calentamiento (8 iteraciones) y se llena la tabla*/
        k = (*idum) / IQ; /* Hacemos la semila más pequeña para que al multiplicarla por IA no nos de problemas de overflow*/
        *idum = IA * (*idum - k * IQ) - IR * k; /*Hemos conseguido aplicar LCG pero usando dos multiplicaciones para evitar overflow en vez de hacer una multiplicacion muy grande*/
        if (*idum < 0) *idum += IM; /* Hacemos que el número esté entre 0 y IM -1 si no lo está*/
        if (j < NTAB) iv[j] = *idum; /* Empezamos a llenar la tabla después de hacer el calentamiento*/
    }
    iy = iv[0]; /* Tomamos el primer valor de la tabla*/
  }
    /*Generación del numero aleatorio*/
    k = (*idum) / IQ;
    *idum = IA * (*idum - k * IQ) - IR * k; /* Hacemos LCG como antes evitando overflow*/
    if (*idum < 0) *idum += IM; /* Hacemos que el número esté entre 0 y IM -1 si no lo está*/
    j = iy / NDIV; /* Convierte el valor iy a un índice entro 0 y NTAB-1 (un índice aleatorio)*/
    iy = iv[j]; /* Obtenemos la nueva semilla para el próximo número*/
    iv[j] = *idum; /* Se actualiza la nueva semilla con la antes generada*/

    if ((temp = AM * iy) > RNMX) return RNMX; else return temp; /* Hacemos la transformación a un número en [0,1) comprobando que no sea 1*/
}
```

Problema de overflow: Al aplicar el algoritmo de Park y Miller hacemos esto:

$$X_{n+1} = (a \cdot X_n) \mod m$$ Donde a es IA y m es IM.

El problema viene con la multiplicación de a*Xn ( semilla usada), esta semilla Puede llegar a ser (2^31) -1, de forma que al multiplicarse por a, supere el límite de long (2^32) y por tanto se produzca overflow. Para evitarlo, se utiliza un método matemático propuesto por Schrage (1983) que permite realizar la operación modular sin generar números grandes.

La idea consiste en reorganizar la operación usando la descomposición del módulo (m). Se calcula (m = a * q + r), con (q = m/a = 127773) y (r = m (mod a) = 2836). Luego, se descompone la semilla X_n = qk + (X_n (mod q)), donde k = X_n/q. Sustituyendo en la ecuación original y aplicando propiedades del módulo, se obtiene una forma equivalente y segura:

a*X_n (mod m) = a*(X_n (mod q)) - r*k

Como esta expresión puede ser negativa, si el resultado es menor que cero se corrige sumando m. De este modo, todas las operaciones intermedias permanecen dentro del rango de enteros normales, menores que el propio módulo m, evitando así cualquier desbordamiento.

Esta técnica aparece explicada en el capítulo 7 (Random Numbers) del libro Numerical Recipes in C.

Tras haber generado un número decimal en [0,1). Podemos adaptarlo a un número en el rango que se nos pide de la siguiente forma:

```
static long seed = 0;
int rango;
float rand;

if(seed == 0){
  seed = -(long)time(NULL); /*Va a ser la semilla inicial*/
}
rango = sup - inf + 1;
rand = ran1(&seed);
return inf + (int)(rand*rango);
```

Donde multiplicamos el decimal por el rango del intervalo, y le sumamos el primer número del intervalo para asegurarnos que esté dentro.

**6.2 Justifica lo más formalmente que puedas la corrección (o dicho de otra manera, el porqué ordena bien) del algoritmo InsertSort.**

El algoritmo InsertSort ordena correctamente un array porque en cada iteración toma el elemento situado en la posición i y lo inserta en el lugar adecuado dentro de un subarray ya ordenado comprendido entre el índice primero y el i-1. De esta manera, el subarray queda ordenado de manera creciente. Durante el proceso, los elementos mayores se desplazan una posición a la derecha hasta encontrar la posición correcta para insertarla. Finalmente, al terminar el bucle, todos los elementos del array entre están ordenados de forma ascendente, demostrando el correcto funcionamiento del algoritmo.

**6.3 ¿Por qué el bucle exterior de InsertSort no actúa sobre el último elemento de la tabla?**

El bucle exterior de InsertSort no actúa sobre el último elemento (en nuestro caso el primer elemento) porque el algoritmo funciona tomando un subarray ya ordenado al que se van insertando los elementos siguientes en su posición ordenada correcta. Entonces, se coge el último elemento como primer elemento de este subarray y no hace falta compararlo porque no hay ningún otro elemento en el subarray.

**6.4 ¿Cuál es la operación básica de InsertSort?**

La operación básica de InsertSort es la comparación. Es decir, el algoritmo toma cada elemento del arreglo a partir del segundo, lo compara con los elementos anteriores y va moviendo hacia la derecha aquellos que son mayores, hasta encontrar el lugar adecuado donde insertar el elemento actual, así hasta dejar el array ordenado.

**6.5 Dar tiempos de ejecución en función del tamaño de entrada n para el caso peor WBS(n) y el caso mejor BBS(n) de InsertSort y BubbleSort. Utilizad la notación asintótica (O, Θ, o, Ω,etc) siempre que se pueda.**

En InsertSort, la operación básica consiste en la comparación de los elementos. En el mejor caso, cuando el arreglo ya está ordenado, cada elemento se compara solo una vez y no se realizan desplazamientos, por lo que el tiempo de ejecución es Θ(n). En el peor caso, cuando el arreglo está ordenado de manera inversa, cada elemento debe moverse hasta el inicio, generando aproximadamente $\frac{n(n-1)}{2}$ comparaciones y desplazamientos, lo que da un tiempo de ejecución de $\Theta(n^2)$.

En BubbleSort, el mejor caso ocurre cuando el arreglo ya está ordenado y se utiliza una versión optimizada que detecta que no hubo intercambios, terminando tras una sola pasada con un tiempo de Θ(n). En el peor caso, con el arreglo en orden inverso, se realizan todos los intercambios posibles y se recorren todas las posiciones, dando también un tiempo de $\Theta(n^2)$.

Ambos algoritmos comparten la misma complejidad cuadrática en el peor caso y lineal en el mejor caso para entradas ya ordenadas.

**6.6 Compara los tiempos medios de reloj, asA como el caso medio, peor y mejor obtenidos para InsertSort y BubbleSort, justifica las similitudes o diferencias entre ambos (es decir, indicad si las gráficas son iguales o distintas y por qué).**

La respuesta a este apartado se ha ido comentando en la explicación de las gráficas, no obstante, haremos un resumen de por qué es así.

Los tiempos medios de reloj de InsertSort y BubbleSort muestran que, aunque ambos crecen según el orden $O(N^2)$, son distintos en su magnitud en la práctica pues la constante de Bubble Sort es ½ y la de InsertSort ¼ . La curva de tiempo de InsertSort es consistentemente más baja que la de BubbleSort, lo que significa que es sustancialmente más rápido. Esta diferencia se debe a que InsertSort tiene un factor constante significativamente menor: InsertSort ejecuta desplazamientos de elementos, que son operaciones más eficientes en el hardware que los costosos intercambios repetitivos y el overhead de BubbleSort.

Por lo tanto, aunque ambos son teóricamente "lentos", InsertSort es la opción más eficiente en términos de tiempo real de ejecución.

# 7. Conclusiones finales.

Después de completar el código en C con las funciones que faltaban, ha sido cuando realmente hemos visto la importancia de esta pŕactica: entender completamente el funcionamiento y el rendimiento de los algoritmos InsertSort y BubbleSort. Durante toda la práctica se ha demostrado experimentalmente todas las ideas teóricas que se habían visto en clase. Se han calculado y comparado los tiempos de reloj medios de ambos algoritmos y su mejor, peor y medio tiempo en OBs. También se implementó un generador de números aleatorios utilizando el algoritmo de Park y Miller y la técnica de Schrage, produciendo una distribución uniforme y equiprobable. Finalmente, el uso de herramientas como Valgrind y GnuPlot aseguró el correcto funcionamiento del código y la claridad de la presentación de los resultados.