Análisis de Algoritmos 2022/2023

Práctica 1

Grosso Christian, Ștefan Gabriel, Grupo 8.

Código	Gráficas	Memoria	Total

1. Introducción.

En esta práctica se desarrollará un conjunto de rutinas en lenguaje C para generar permutaciones aleatorias de un número determinado de elementos. Estas rutinas se utilizarán posteriormente como entradas para implementar y probar algoritmos de ordenación. A su vez, se crearán funciones que medirán el tiempo de ejecución de los algoritmos de ordenación usando las permutaciones generadas.

2. Objetivos

2.1 Apartado 1

Implementar una rutina que genere un número aleatorio equiprobable entre dos valores **inf** y **sup** dados, incluidos ambos límites como parámetros.

2.2 Apartado 2

Desarrollar una rutina que, al recibir un valor **N**, devuelva una lista con una permutación aleatoria de tamaño **N**, cuyos elementos sean los números del 1 al **N**.

2.3 Apartado 3

Implementar una rutina que, al recibir como entrada un valor **N** y un valor **n_perms,** devuelva una lista que contenga **n_perms** permutaciones aleatorias, cada una de tamaño **N**, generadas utilizando la rutina del Apartado 2.

2.4 Apartado 4

Crear una rutina que implemente el algoritmo de ordenación *Bubble Sort* (ordenación por intercambio de burbujas) sobre una permutación. La rutina deberá recibir como entradas un **array** que contiene la permutación y los índices del primer (**ip**) y último (**iu**) elemento de la permutación. Devolverá el número de veces que se ha ejecutado la operación básica (OB).

2.5 Apartado 5

Crear una rutina que, al recibir como entradas una función de ordenación **method**, una cadena de caracteres file, un número mínimo **num_min**, un número máximo **num_max**, un incremento **incr** y un número **n_perms**, genere **n_perms** permutaciones cuyos tamaños varíen de **num_min** a **num_max** con un incremento **incr**.

Crear una rutina que, al recibir como entradas una función de ordenación **method**, el número de permutaciones **n_perms**, el tamaño de cada permutación **N** y una estructura para guardar datos (**ptime**), mida el tiempo de ejecución del algoritmo de ordenación y el número de operaciones básicas realizadas. Todos estos datos se almacenarán en la estructura **ptime**.

Crear una rutina que, al recibir como entradas una cadena de caracteres **file**, una estructura **ptime** y un número **n_times**, escriba en el fichero **file** los datos contenidos—— en la estructura **ptime**.

2.6 Apartado 6

Implementar una mejora del algoritmo *Bubble Sort*, denominado *BubbleSortFlag*. Esta rutina tendrá las mismas entradas que la del Apartado 4, pero con la optimización de detener la ejecución si no se realizan intercambios en una pasada. Devolverá también el número de veces que se ejecuta la operación básica (OB).

3. Herramientas y metodología

El ambiente de desarrollo utilizado es el Subsistema de Windows para Linux (**WSL**). Otras herramientas utilizadas para el desarrollo del código fueron **Visual Studio Code** y la **extensión GCC** para Visual Studio Code. Para la parte de los gráficos, utilicé **Gnuplot** con las guías disponibles en Moodle. Las pruebas del código fueron principalmente las proporcionadas por el profesor en el fichero "**MAKEFILE**", en algunos casos modificando los parámetros para verificar los casos límite y el retorno de los errores.

3.1 Apartado 1

Para la generación de números aleatorios, inicialmente utilicé un algoritmo que empleaba bits de orden bajo, basándome en mis conocimientos previos. Posteriormente, opté por implementar una solución que genera números aleatorios utilizando bits de orden alto, tomando una idea del capítulo 7 ("Random Numbers") del libro "Numerical Recipes in C: The Art of Scientific Computing" para favorecer una distribución uniforme de los valores generados.

3.2 Apartado 2

Para la generación de las permutaciones, implementé el algoritmo traduciendo el pseudocódigo presente en el texto de la práctica (Fisher-Yates), asignando a cada celda del array el valor de *i*, y luego intercambiando la celda array[i] con la celda array[random_num(i, N)] para cada *i*. Dentro de la misma función se realiza la asignación de memoria dinámica, y dicha memoria se libera en caso de error. Posteriormente, modifiqué el primer ciclo donde ocurre la asignación, de forma que se asignara a cada celda el valor de $i + 1$, para que los valores fueran de 1 a N en lugar de 0 a N-1.

3.3 Apartado 3

Para la generación del array de permutaciones, se asigna memoria dinámica, y para cada celda del array se llama a la función del Apartado 3, que a su vez asigna memoria. La memoria se libera en caso de error.

3.4 Apartado 4

Para la función de *Bubble Sort*, se implementó el algoritmo visto en las clases teóricas. Básicamente, el algoritmo ordena una lista comparando dos elementos vecinos y, si están desordenados, los intercambia. Repite este proceso varias veces, pasando por toda la lista, hasta que ya no hay más elementos por intercambiar. Al final de cada pasada, el elemento más grande "sube" a su posición correcta, como si fuera una burbuja que asciende. Se repite hasta que toda la lista esté ordenada.

3.5 Apartado 5

Para la función *average_sorting_time*, inicialmente se medía el tiempo usando la función clock(), pero, dado que esta función tiene poca precisión, fue sustituida por la función gettimeofday() de la librería <sys/time.h>. Esta función proporciona una mayor precisión, ya que trabaja con la estructura timeval, que tiene dos campos: tv_sec (que contiene el tiempo en segundos desde el 01/01/1970) y tv_usec (que contiene los microsegundos). Básicamente, se llama a la función para guardar el tiempo actual, se realiza la medición del ordenamiento, se recopilan los datos relacionados con la operación básica y luego se vuelve a guardar el tiempo. Restando el segundo tiempo al primero, obtenemos el tiempo de ejecución del algoritmo.

En la función *generate_sorting_time*, se asigna memoria para la estructura que almacenará los datos sobre los tiempos y la ejecución de la operación básica. Además, con los valores de mínimo, máximo e incremento, se calcula cuántas veces se repetirá el algoritmo y para qué tamaños de datos.

En la función *save_time_table*, los valores de la estructura que contiene los tiempos y las estadísticas sobre la ejecución de la operación básica se guardan en un archivo utilizando la función fprintf de la librería <stdio.h>.

3.6 Apartado 6

Para la función de *Bubble Sort* con bandera, se implementó el algoritmo visto en las clases teóricas. Básicamente, el algoritmo *Bubble Sort* con bandera (*Bubble Sort Flag*) es una versión mejorada del algoritmo de la burbuja. La diferencia principal es que incluye una "bandera" (un indicador) que ayuda a detener el algoritmo si en una pasada completa no se hizo ningún intercambio. Esto significa que la lista ya está ordenada y no es necesario seguir haciendo más pasadas.

4. Código fuente

4.1 Apartado 1

```
Function: random_num
Rutine that generates a random number between two given numbers
Input:
    -int inf: lower limit
    -int sup: upper limit
Output:
    -int: random number
int random_num(int inf, int sup)
{
  if(inf < 0  || sup < 0 || sup < inf) return ERR; //Parameters Check

  int c=sup-inf;
  int rd_num;
  rd_num = inf+(int)((float)c*rand()/(RAND_MAX+1.0)); //Inclusive

  return rd_num;
}
```

4.2 Apartado 2

```
Function: generate_perm
Rutine that generates a random permutation
Input:
    -int n: number of elements in the permutation
Output:
    -int *: pointer to integer array that contains the permutation or NULL in case of
            error
int* generate_perm(int N)
{
  if(N <= 0) return NULL;//Parameters Check
  int i, t, r;
  int*perm=(int*)malloc(sizeof(int)*N);
  if(perm == NULL) return NULL;//Malloc check
  for(i=0;i<N;i++)
    perm[i]=i+1;
  for(i=0;i<N;i++){
    t=perm[i];
    r=random_num(0,N-1);
    if (r == ERR){ //Random Number Check
      free(perm); //Free memory
      return NULL;
    }
    perm[i]=perm[r];
    perm[r]=t;
  }
  return perm;
}
```

4.3 Apartado 3

```
Function: generate_permutations
Routine that generates n_perms random permutations with N
Input:
    -int n_perms: Number of permutations
    -int N: Number of elements in each permutation
Output:
    -int**: Array of pointers to integer that point to each of the permutations
            NULL in case of error
int** generate_permutations(int n_perms, int N)
{
  if(n_perms<1 || N<1) return ERR; //Parameters check

  int**permArray=(int**)malloc(sizeof(int*)*n_perms);

  if(permArray == NULL) return ERR; //Malloc Check

  int i;
  for(i=0;i<n_perms;i++){
    permArray[i]=generate_perm(N);
    if (permArray[i]==ERR){ //Gen Perm Check
      free(permArray); //Free memory
      return ERR;
    }
  }
  return permArray;
}
```

4.4 Apartado 4

```
Function: BubbleSort
Routine which implements the bubblesort algorithm
Input:
    -int* array: array pointer
    -int ip: first array index
    -int iu: last array index
Output:
    -int: Swaps Counter or -1 in case of Error
int BubbleSort(int* array, int ip, int iu){
    if (array == NULL || ip < 0 || iu < ip) return ERR; //Parameters check
    int count = 0;
    int temp,a;
    for (int i = ip; i < iu; i++) {
        a = iu - (i - ip);
        for (int j = ip; j < a; j++) {
            count++;
            if (array[j] > array[j + 1]) {
                temp = array[j];
                array[j] = array[j + 1];
                array[j + 1] = temp;
            }
        }
    }
    return count;
}
```

4.5 Apartado 5

```
Function: generate_sorting_times
Routine which calculates how much times have to execute the time measure, and allocate
memory for time_aa structure.
Inputs:
    -pfunc_sort method: sorting method
    -char* file: string with the name of the file where the output will be saved
    -int num_min: min size of the permutation
    -int num_max: max size of the permutation
    -int incr: incrementation for the permutation size
    -int n_perms: number of permutations for each size
Outputs: short OK (!ERR) if there are not errors, ERR (-1) if case of error
short generate_sorting_times(pfunc_sort method, char* file, int num_min, int num_max,
                             int incr, int n_perms)
{
  int n_times = (num_max - num_min) / incr + 1; //num min included
  int ret,N;
    PTIME_AA time_aa = (PTIME_AA) malloc(n_times * sizeof(TIME_AA));
    if (time_aa == NULL) return ERR;
    for (int i = 0; i < n_times; i++) {
        N=num_min+(incr*i);
        ret=average_sorting_time(method, n_perms, N, &time_aa[i]);
        if(ret == ERR){ //Return check
          free(time_aa); //free memory
          return ERR;
        }
    }
    if(save_time_table(file,time_aa,n_times)==ERR) return ERR; //Return Check

    return OK;
}
```

```
Function: average_sorting_time
Routine which calculate:
    - the avg exe time
    - max min and avg basic operation executed
of the “metodo” function on “n_perms” permutations, each of N elements and put all the
data in ptime structure
Inputs:
    -pfunc_sort metodo: function to be measured
    -int n_perms: number of permutations to be analyzed
    -int N: number of elements for each permutation
    -PTIME_AA ptime: Pointer to struct who contains results
Output:
    -int OK (!ERR) if there are not errors, ERR (-1) in case of error

short average_sorting_time(pfunc_sort metodo, int n_perms, int N, PTIME_AA ptime)
{
  if(n_perms <= 0 || N <= 0 || ptime == NULL)return ERR; //Parameters check
  int i,j;
  double t1,t2,contT=0;
  int contOB=0,averageOB=0,minOB=2147483647,maxOB=0;
  int**permArray=generate_permutations(n_perms, N);
  if(permArray == NULL) return ERR; //Malloc check
  struct timeval tv1,tv2;
  gettimeofday(&tv1,NULL);
  for(i=0;i<n_perms;i++){
    j=metodo(permArray[i],0,N-1);
    if(j==ERR) return ERR; //return Check
    //OB Operations
    if(j<minOB) minOB=j;
    if(j>maxOB) maxOB=j;
    contOB+=j;
  }
  gettimeofday(&tv2,NULL);
  t1 = tv1.tv_sec*1000 + (tv1.tv_usec / 1000.0); //in ms
  t2 = (tv2.tv_sec*1000 + (tv2.tv_usec / 1000.0) - t1); //in ms
  contT=t2/n_perms;
  averageOB=contOB/n_perms;

  //saving data in struct
  ptime->n_elems=n_perms;
  ptime->N=N;
  ptime->time=contT;
  ptime->average_ob=averageOB;
  ptime->max_ob=maxOB;
  ptime->min_ob=minOB;
  return OK;
}
```

```
Function: save_time_table
Routine which save in a file a table with all the array of struct data.
Inputs:
    -char* file: string with the name of the file
    -PTIME_AA time_aa: array of struct where there are the data to be write
    -n_times: number of struct in the struct array
Output:
    -int OK (!ERR) if there are not errors, ERR (-1) in case of error

short save_time_table(char* file, PTIME_AA ptime, int n_times)
{
  FILE *f = fopen(file, "w");
    if (!f) return ERR; //File Check


    fprintf(f, "Size\tTime\t\tAvg OB\tMax OB\tMin OB\n");

    // Writing Data
    for(int i = 0; i<n_times;i++){
      fprintf(f, "%d\t%.6f\t%.2f\t%d\t%d\n", ptime[i].N, ptime[i].time,
            ptime[i].average_ob, ptime[i].max_ob, ptime[i].min_ob);
      }
    fclose(f);
    return OK;
}
```

4.6 Apartado 6

```
Function: BubbleSort
Routine which implements the bubblesort algorithm with flag
Input:
    -int* array: array pointer
    -int ip: first array index
    -int iu: last array index
Output:
    -int: Swaps Counter or -1 in case of Error
int BubbleSortFlag(int* array, int ip, int iu)
{
    if (array == NULL || ip < 0 || iu < ip) return ERR; //Parameters Check
    int swapped; //Flag
    int i, j;
    int cont = 0;
    for (i = iu; i > ip; i--) {
        swapped = 0;
        for (j = ip; j < i; j++) {
            cont++;
            if (array[j] > array[j+1]) {
                int temp = array[j];
                array[j] = array[j+1];
                array[j+1] = temp;
                swapped = 1; //Flag update
            }
        }
        if (!swapped) return cont;
    }
    return cont;
}
```

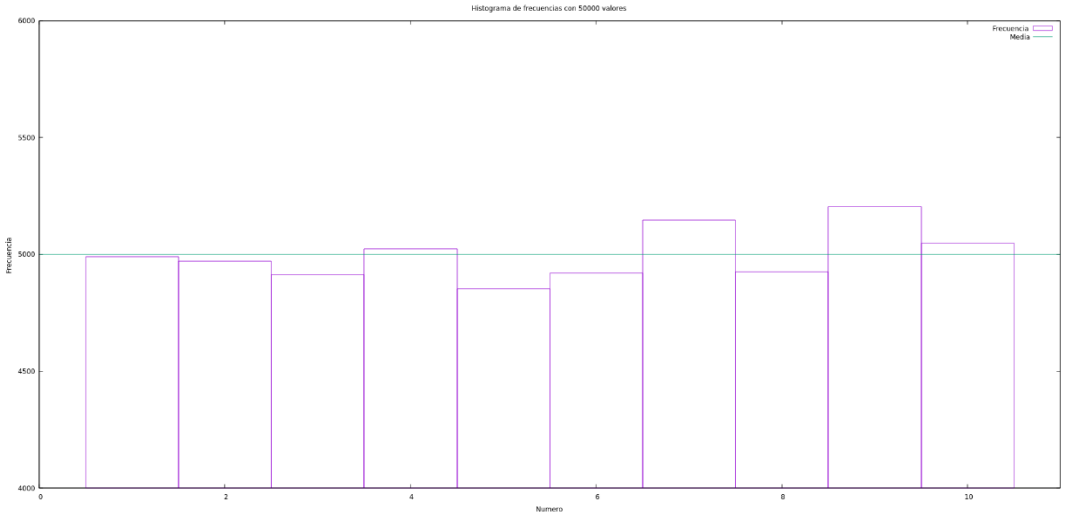
5. Resultados, Gráficas

5.1 Apartado 1

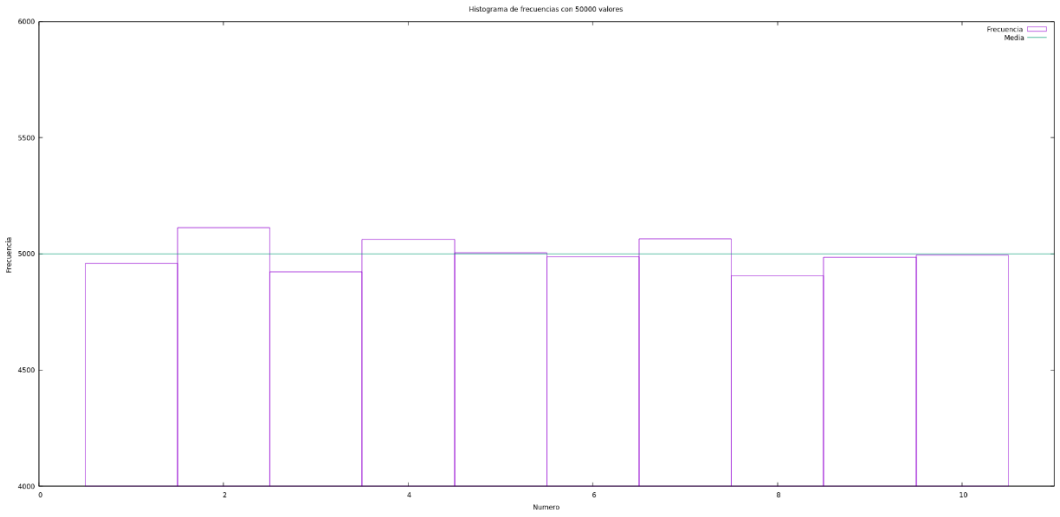
Cuando se comparan dos algoritmos para la generación de números aleatorios, es crucial analizar su calidad en términos de aleatoriedad y eficiencia. Un buen algoritmo de generación de números aleatorios debe producir secuencias que parezcan impredecibles y distribuidas uniformemente.

A continuación, se presentan los resultados de la generación de números aleatorios con 50000 valores.

Algoritmo que emplea bits de orden Bajo: Error: 1.69%



Algoritmo que emplea bits de orden Alto: Error: 0.97%



El primero utiliza un algoritmo que emplea los **bits de orden bajo**, es decir, los menos significativos, y presenta un MAPE (Mean Absolute Percentage Error) del 1.69%, mientras que el segundo utiliza los **bits de orden alto**, o los más significativos, y presenta un MAPE de 0.97%.

5.2 Apartado 2

Este algoritmo sigue el método de permutación de Fisher-Yates, que garantiza que cada permutación tiene la misma probabilidad de ser generada.

5.3 Apartado 3

Esta rutina nos devuelve el array de permutaciones generadas.

5.4 Apartado 4

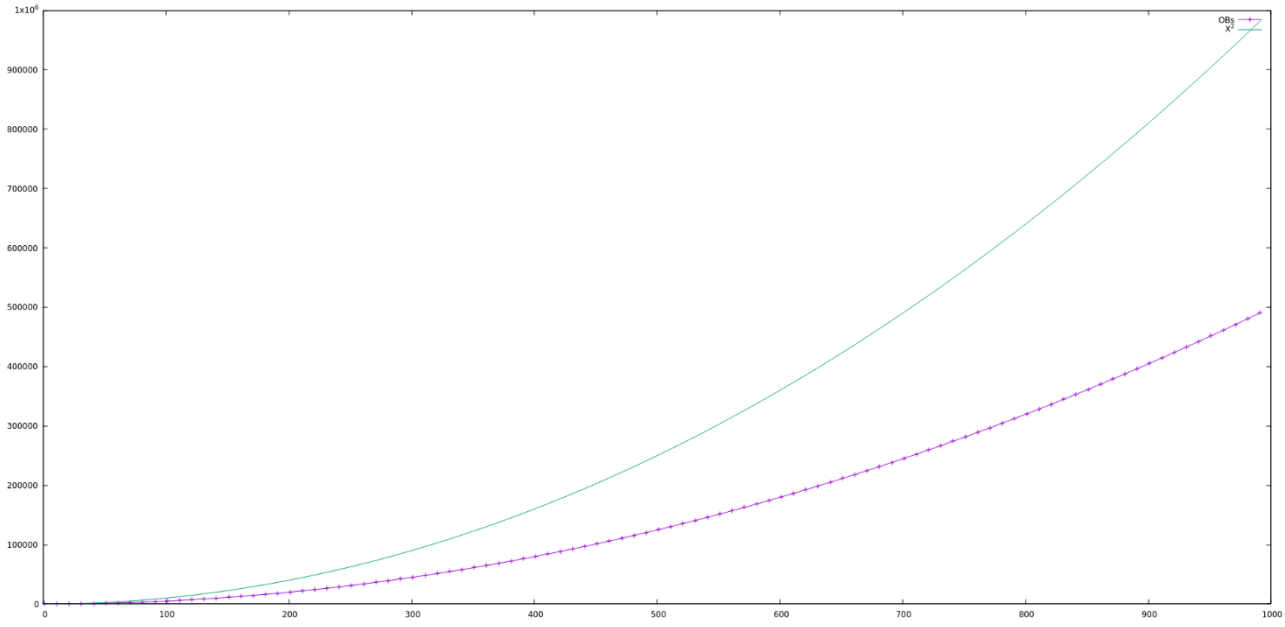
El algoritmo bubble sort es un algoritmo de ordenamiento estable y que no requiere el uso de memoria adicional (in-place). No se considera un algoritmo eficiente porque realiza muchas operaciones superfluas, especialmente si el arreglo está casi ordenado, ya que sigue haciendo comparaciones incluso si los elementos ya están en la posición correcta.

5.5 Apartado 5

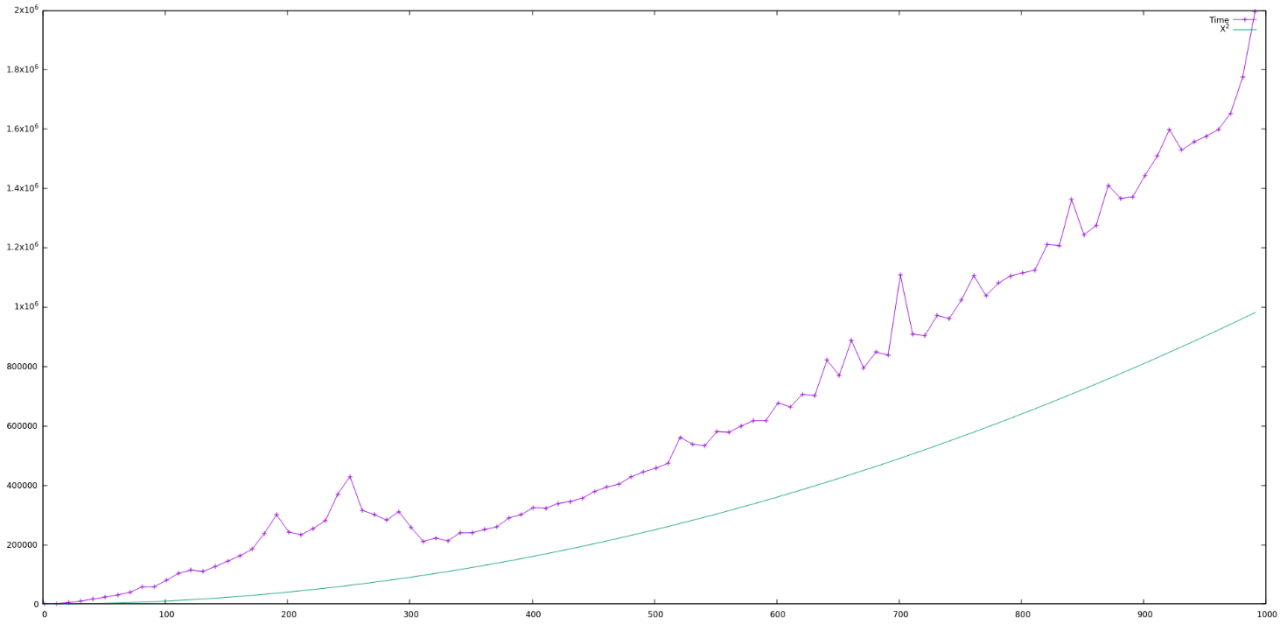
A medida que aumenta el tamaño N de las permutaciones, el tiempo de ejecución crece rápidamente. Esto es consistente con la complejidad temporal del algoritmo bubble sort, que es O(N^2).

De manera similar, el número de veces que se ejecuta la operación básica (comparación entre elementos) también crece cuadráticamente con el tamaño N. Esto refleja la ineficiencia de bubble sort en conjuntos de datos más grandes.

A continuación, se presentan los resultados de la medida del tiempo y del número de operación básicas.



Operaciones Básicas: aumentan al aumentar del tamaño de la permutación siguiendo un andamiento cuadrático.

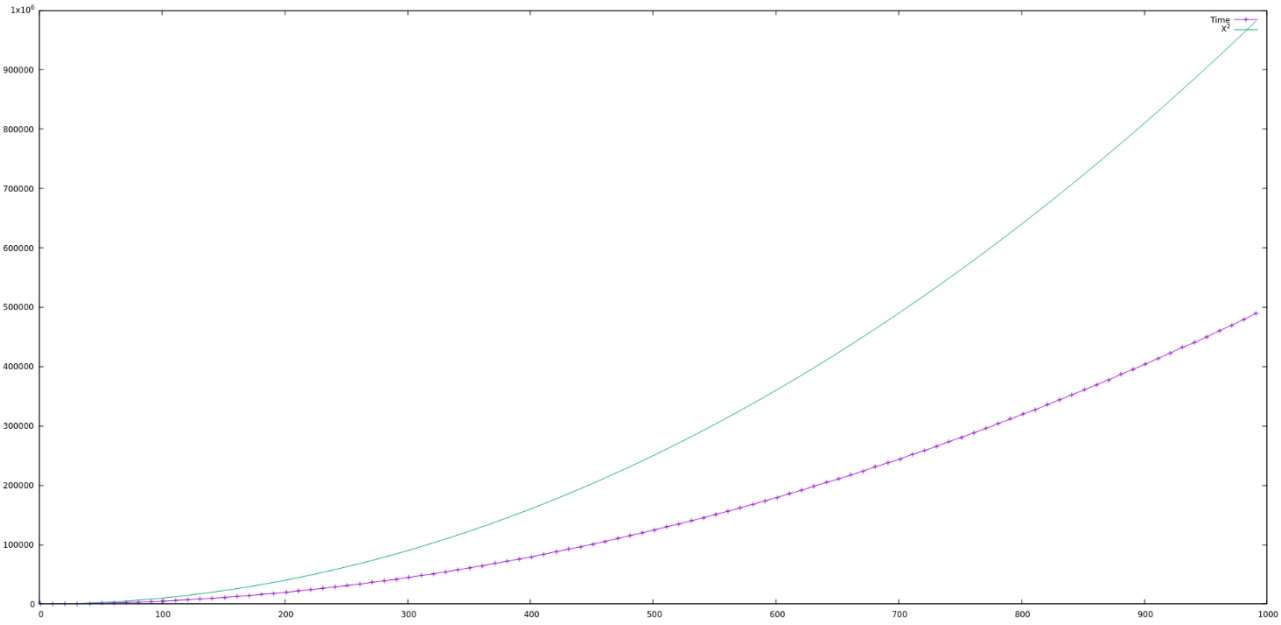


Tiempo: también el tiempo tiene andamiento cuadrático pero el grafico no es tan limpio como el de las operaciones básicas.

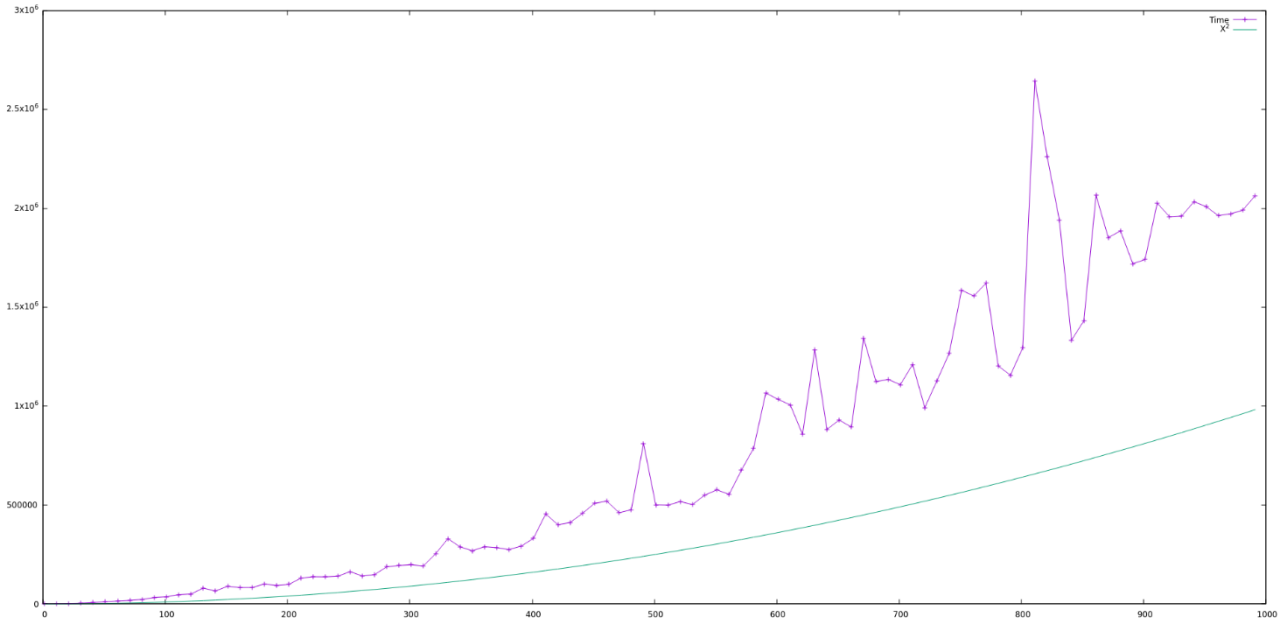
5.6 Apartado 6

El BubbleSortFlag nos da resultados mejores, en el caso mejor tiene complejidad temporal O(n) porque se detiene si no hay intercambio.

A continuación, se presentan los resultados de la medida del tiempo y del número de operación básicas con BubbleSortFlag



Operaciones Básicas: aumentan al aumentar del tamaño de la permutación siguiendo un andamiento cuadrático.



Tiempo: también el tiempo tiene andamiento cuadrático pero el grafico no es tan limpio como el de las operaciones básicas.

6. Respuesta a las preguntas teóricas.

Aquí respondéis a las preguntas teóricas que se os han planteado en la práctica.

6.1 Pregunta 1

Para la generación de números aleatorios, inicialmente utilicé un algoritmo que empleaba bits de orden bajo, basándome en mis conocimientos previos. Posteriormente, opté por implementar una solución que genera números aleatorios utilizando bits de orden alto, tomando una idea del capítulo 7 ("Random Numbers") del libro "Numerical Recipes in C: The Art of Scientific Computing" para favorecer una distribución uniforme de los valores generados.

6.2 Pregunta 2

El algoritmo Bubble Sort ordena correctamente porque:

1. Invariante del bucle: En cada pasada, el mayor de los elementos no ordenados "burbujea" hacia su posición final. Al final de cada iteración, un elemento adicional queda en su lugar correcto.
2. Terminación: Después de n−1n-1n−1 pasadas, todos los elementos están en sus posiciones correctas, ya que cada pasada asegura que un elemento más se ordena.

Por lo tanto, Bubble Sort siempre deja el array completamente ordenado al finalizar.

6.3 Pregunta 3

El bucle exterior de Bubble Sort no actúa sobre el primer elemento en las últimas iteraciones porque, en cada pasada, el mayor elemento no ordenado se mueve a su posición correcta al final del array. Después de cada iteración, el último elemento ya está en su lugar y no necesita ser procesado nuevamente.

6.4 Pregunta 4

La operación básica de Bubble Sort es la comparación entre dos elementos adyacentes del array. Esta operación es fundamental porque determina si es necesario realizar un intercambio para mover los elementos en el orden correcto. Es elegida como operación básica porque es la que mas viene ejecutada y define el algoritmo.

6.5 Pregunta 5

Los tiempos de ejecución de Bubble Sort en notación asintótica son:

Caso peor $W_{BS(}n)$: Ocurre cuando el array está en orden inverso, y el algoritmo realiza el máximo número de comparaciones e intercambios.

- Bubble Sort: $W_{BS}(n)=O(n^2)$
- Bubble Sort con Flag: $W_{BS}(n)=O(n^2)$

Caso mejor (BBS(n)): Ocurre cuando el array ya está ordenado.

- Bubble Sort: $B_{BS}(n)=O(n^2)$, porque sigue recorriendo todo el array aunque no haya intercambios.
- Bubble Sort con Flag: $B_{BS}(n)=O(n)$, ya que el algoritmo detecta que no se necesitan intercambios y termina después de una pasada.

6.5 Pregunta 6

Al comparar los tiempos medios de reloj obtenidos para BubbleSort y BubbleSortFlag, es probable observar lo siguiente:

En el caso peor (cuando el array está completamente desordenado), ambos algoritmos presentan un rendimiento similar. Esto se debe a que en este caso ambos algoritmos recorren el array completo y realizan el máximo número de comparaciones e intercambios, lo que los lleva a un tiempo de ejecución de $O(n^2)$

En el caso mejor (cuando el array está ordenado), BubbleSort sigue realizando n^2 comparaciones, mientras que BubbleSortFlag detecta rápidamente que el array ya está ordenado y finaliza tras la primera pasada. Esto hace que BubbleSortFlag tenga un tiempo de ejecución mucho menor en este caso O(n), mientras que **BubbleSort** mantiene $O(n^2)$

En resumen, las diferencias entre los tiempos se explican por la capacidad de BubbleSortFlag para detectar la ordenación temprana, mientras que BubbleSort sigue recorriendo el array sin optimización.

7. Conclusiones finales.

En esta práctica se implementaron diferentes algoritmos y funciones para la generación de permutaciones aleatorias y la ejecución de algoritmos de ordenación, específicamente BubbleSort y su variante con flag (BubbleSortFlag), con el objetivo de estudiar sus tiempos de ejecución y comportamiento.

Los principales puntos son:

Generación de permutaciones:

- El uso de la función random_num nos permitió generar permutaciones aleatorias eficientemente, lo cual fue crucial para obtener entradas aleatorias para los algoritmos de ordenación. El pseudocódigo utilizado permitió generar permutaciones equiprobables.

BubbleSort y BubbleSortFlag:

- Se implementaron ambos algoritmos y se compararon los tiempos de ejecución en diferentes casos (mejor, peor y promedio).
- En el caso peor (cuando el array está en orden inverso), tanto BubbleSort como BubbleSortFlag tienen un comportamiento similar, ya que ambos realizan la máxima cantidad de comparaciones e intercambios.
- En el caso mejor (cuando el array está ya ordenado), BubbleSortFlag fue significativamente más eficiente al detenerse después de la primera pasada, logrando un tiempo de O(n), mientras que BubbleSort siguió recorriendo el array completo sin optimización.

Comparación de tiempos:

- Los tiempos medios obtenidos para BubbleSortFlag son, en general, más rápidos que para BubbleSort, especialmente en los casos donde el array estaba casi ordenado. Esto se debe a que el flag permite detener el algoritmo si no se requiere realizar más intercambios, ahorrando tiempo de ejecución.
- En arrays completamente desordenados, ambos algoritmos presentan tiempos similares, ya que ambos deben hacer comparaciones e intercambios en cada iteración.

La práctica fue una oportunidad para comprender en profundidad el funcionamiento de los algoritmos de ordenación, en especial BubbleSort y sus variantes. Los resultados obtenidos confirman la ineficiencia de BubbleSort en comparación con otros algoritmos más avanzados, pero también permiten apreciar las optimizaciones simples como el uso de un flag para mejorar el rendimiento.