Análisis de Algoritmos 2022/2023

Práctica 1

Luis Núñez Fernández, Diego José Ruz Estrada, Grupo 1261.

Código	Gráficas	Memoria	Total

1. Introducción.

En esta práctica se ha puesto el objetivo de realizar el estudio de algoritmos relacionados a un conjunto de datos generado de manera aleatoria (números y permutaciones). Para ello hemos desarrollado varias implementaciones, un generador de números aleatorios, un generador de permutaciones aleatorias y otro de n números de permutaciones además de la ejecución de varios algoritmos sobre estas permutaciones con el fin de medir su rendimiento.

Para realizar el análisis usaremos una función que mida el tiempo de ejecución. Con el objetivo de poder hacer una verdadera comparación con diferentes escenarios y posibilidades se realizará una prueba ejecutando la función una cantidad de veces significativa, minimizando así la variabilidad de las pruebas y tiempos de ejecución.

Esta práctica nos permitirá comprender el coste del algoritmo que vamos a utilizar además de proponer posibles cambios para mejorar la eficiencia de la implementación que vamos a realizar.

2. Objetivos

Aquí indicáis el trabajo que vais a realizar en cada apartado.

2.1 Apartado 1

En este apartado se busca realizar una función que genere números aleatorios equiprobables entre dos enteros dados (ambos incluidos), para ello haremos uso de la rutina de C rand de la librería stdlib. Para comprobar la equiprobabilidad usaermos un histograma creado a través de un volcado de texto desde nuestro programa.

2.2 Apartado 2

En este segundo apartado se nos proporciona un pseudocódigo de un algoritmo que genera permutaciones aleatorias utilizando a su vez la función que hemos desarrollado en el apartado anterior.

2.3 Apartado 3

Este apartado comprende a la primera parte del Bloque 2, se trata de desarrollar una rutina que genere n números de permutaciones equiprobables, de nuevo, usaremos la función creada en el anterior apartado.

2.4 Apartado 4

En el apartado 4 se pide crear una función (BubbleSort) para el método por ascenso de la burbuja.

2.5 Apartado 5

El objetivo es implementar un conjunto de funciones que permitan medir y almacenar los tiempos de ejecución y el número de operaciones básicas del algoritmo de ordenamiento BubbleSort sobre varias permutaciones. Estos tiempos se guardan en una estructura y se pueden registrar en un archivo. Las funciones medirán el rendimiento promedio, mínimo y máximo de las ejecuciones para diferentes tamaños de entrada y números de permutaciones.

2.6 Apartado 6

En este último apartado el objetivo es mejorar el algoritmo BubbleSort añadiendo un *flag* que detiene la ejecución cuando la lista ya está ordenada. Para ello se debe implementar una función BubbleSortFlag que utilice este mecanismo y luego comparar su rendimiento con la versión original de *BubbleSort* en los diferentes casos.

3. Herramientas y metodología

Durante toda la práctica hemos usado el entorno Linux junto VSCode, además hemos hecho uso de la herramienta Valgrind para controlar posibles errores. Por otro lado, para realizar las gráficas hemos utilizado Gnuplot.

4. Código fuente

4.1 Apartado 1

```
int random_num(int inf, int sup)
{
if (inf > sup || inf < 0 || sup < 1){
return -1;
}
return rand() % (sup - inf) + inf;
}
```

4.2 Apartado 2

```
int* generate_perm(int N)
{
int *perm;

int i;
int cambio;
int ind;
int aux;

perm = (int*)malloc(N * sizeof(int));
if (perm == NULL){
return NULL;
}

for (i=0; i<N; i++){
perm[i] = i;
}

for (i=0; i<N; i++){
cambio = perm[i];
aux = random_num(i, N);
if (aux == -1){
free(perm);
return NULL;
```

```
}
ind = aux;
perm[i] = perm[ind];
perm[ind] = cambio;
}

return perm;
}
```

4.3 Apartado 3

```
int** generate_permutations(int n_perms, int N) {
int i, j;
int *aux;

int **permutations = malloc(n_perms * sizeof(int*));
if (permutations == NULL) {
return NULL;
}

for (i = 0; i < n_perms; i++) {
aux = generate_perm(N);
if (aux == NULL){
for (j=0; j<i; j++){
free(permutations[j]);
}
free(permutations);
return NULL;
}
permutations[i] = aux;
}

return permutations;
}
```

4.4 Apartado 4

```
int BubbleSort(int* array, int ip, int iu)
{
int i, j, temp;
int count = 0;

if (!array || ip < 0 || iu < 0 || ip > iu) return ERR;
```

```
for (i = ip; i < iu; i++) {
for (j = iu; j > i; j--) {
if (array[j] < array[j - 1]) {
temp = array[j];
array[j] = array[j - 1];
array[j - 1] = temp;
count++;
}
}
}
return count;
}
```

4.5 Apartado 5

```
short average_sorting_time(pfunc_sort metodo, int n_perms,int N, PTIME_AA ptime)
{
int i, min=__INT_MAX__, max=0, sumOB=0, err_aux;
float sumT=0;
int **perms=NULL, *obPerms;
clock_t start, end;
double final = 0;

if (metodo == NULL || n_perms < 1 || N < 1 || ptime == NULL){
return ERR;
}

obPerms = (int*) malloc(n_perms * sizeof(int));
if (obPerms == NULL){
printf("Error al asignar memoria para las permutaciones");
return 1;
}

ptime->N = N;

ptime->n_elems = n_perms;

perms = generate_permutations(n_perms, N);
if (perms == NULL){
printf("ERROR en generate_permutations\n");
return ERR;
}

for (i=0; i<n_perms; i++){
start = clock();
```

```
err_aux = metodo(perms[i], 0, N - 1);
if (err_aux == ERR){
return ERR;
}
obPerms[i] = err_aux;

end = clock();

final = ((double)(end - start)) / CLOCKS_PER_SEC;
sumT += final;
sumOB += obPerms[i];

if (obPerms[i] < min){
min = obPerms[i];
}
if (obPerms[i] > max){
max = obPerms[i];
}
}

ptime->min_ob = min;

ptime->max_ob = max;

ptime->time = sumT/n_perms;

ptime->average_ob = sumOB/n_perms;

for (i=0; i<n_perms; i++){
free(perms[i]);
}
free(perms);
free(obPerms);

return OK;
}
```

```
short generate_sorting_times(pfunc_sort method, char* file, int num_min, int num_max, int
incr, int n_perms)
{
TIME_AA *pt = NULL;
int i, arr_tam, cont=0;
```

```
if (method == NULL || file == NULL || num_min < 1 || num_max < 1 || num_min > num_max ||
incr < 1 || n_perms < 1){
return ERR;
}
cont = num_min;

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

pt = (TIME_AA*) malloc(arr_tam * sizeof(TIME_AA));
if (pt == NULL){
return ERR;
}

for (i=0; i<arr_tam; i++){
if (average_sorting_time(method, n_perms, cont, &pt[i]) == ERR){
free(pt);
return ERR;
}
cont+=incr;
}

if (save_time_table(file, pt, arr_tam) == ERR){
free(pt);
return ERR;
}

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

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

if (file == NULL || !ptime || n_times < 1){
return ERR;
}

f = fopen(file, "w");
if (f == NULL){
return ERR;
}

for (i=0; i<n_times; i++){
```

```
if (fprintf(f, "%d %f %f %d %d\n", ptime[i].N, ptime[i].time, ptime[i].average_ob,
ptime[i].max_ob, ptime[i].min_ob) < 0){
fclose(f);
return ERR;
}
}

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

4.6 Apartado 6

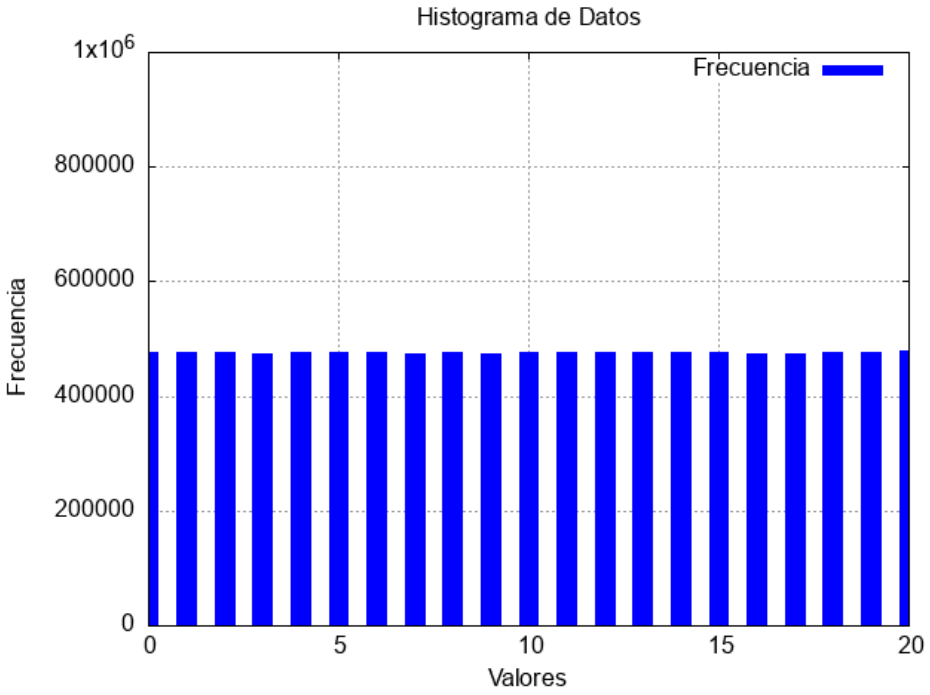
```
int BubbleSortFlag(int* array, int ip, int iu)
{
int i, j, temp, flag;
int count = 0;

if (!array || ip < 0 || iu < 0 || ip > iu) return ERR;

for (i = ip; i < iu; i++) {
flag = 0;
for (j = iu; j > i; j--) {
if (array[j] < array[j - 1]) {
temp = array[j];
array[j] = array[j - 1];
array[j - 1] = temp;
count++;
flag = 1;
}
}

if (flag == 0){
break;
}
}

return count;
}
```

5. Resultados, Gráficas

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

5.1 Apartado 1

```
diego@diego-Victus-by-HP-Laptop-16-e0xxx:~/Documentos/universidad/anal/AALG-main$ ./exercise1 -limInf 1 -limSup 100 -numN 5
Practice no 1, Section 1
Done by: Luis Nuñez & Diego Ruz
Grupo: 1261
66
51
18
68
40
```



Se puede observar en el histograma como la generación de números es equiprobable

5.2 Apartado 2

```
diego@diego-Victus-by-HP-Laptop-16-e0xxx:~/Documentos/universidad/anal/AALG-main$ ./exercise2 -size 8 -numP 8
Practice number 1, section 2
Done by: your names
Group: Your group
1 6 3 2 0 7 4 5
5 4 7 3 1 0 2 6
6 2 4 7 1 3 0 5
7 6 1 2 0 3 5 4
0 6 2 3 1 4 7 5
4 2 0 6 3 1 5 7
1 6 2 7 5 3 0 4
5 7 0 4 6 1 2 3
```

La equiprobabilidad de las permutaciones se evalúa con la de los números.

5.3 Apartado 3

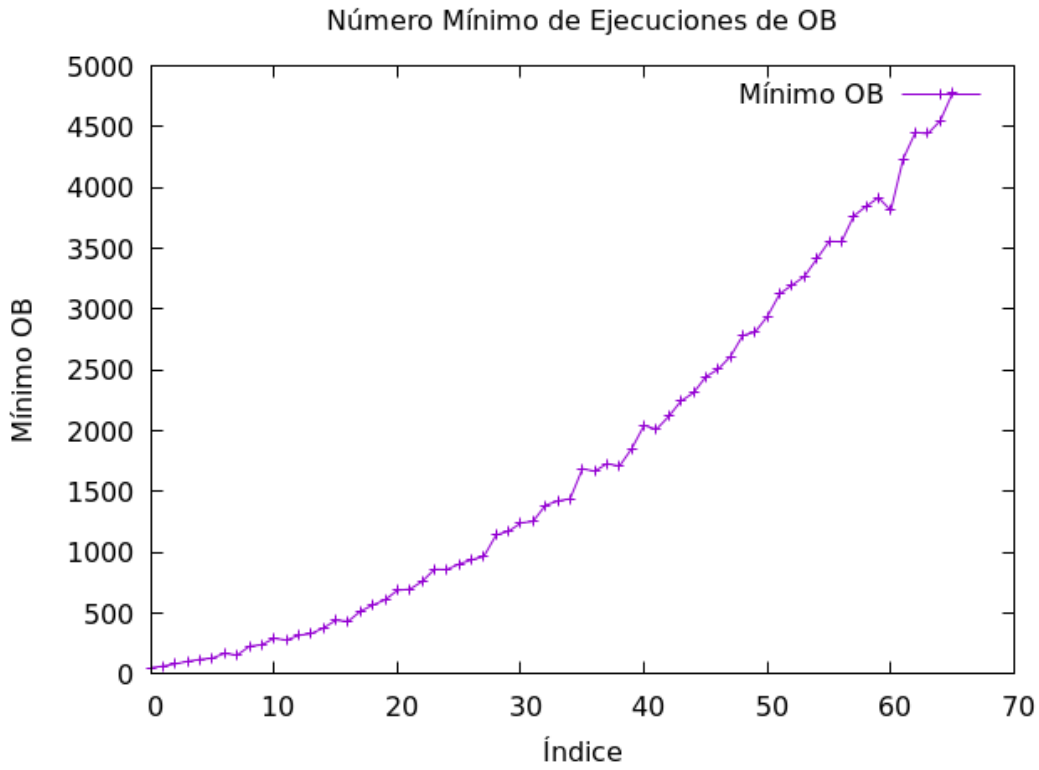
```
diego@diego-Victus-by-HP-Laptop-16-e0xxx:~/Documentos/universidad/anal/AALG-main$ ./exercise3 -size 10 -numP 15
Practice number 1, section 3
Done by: your names
Group: Your group
3 2 1 6 4 0 5 9 8 7
3 5 9 6 1 2 7 8 0 4
8 9 6 5 2 0 4 7 1 3
1 2 0 9 6 7 5 3 4 8
3 9 6 5 7 0 2 8 4 1
7 9 4 6 1 3 2 0 5 8
4 2 7 3 1 5 6 0 8 9
8 6 9 2 7 4 3 1 5 0
9 7 4 2 8 1 6 3 5 0
1 0 2 7 4 9 6 8 3 5
3 7 2 0 9 1 5 6 8 4
2 5 4 3 0 7 1 8 9 6
2 7 5 3 4 8 6 1 9 0
8 3 6 2 0 9 4 1 7 5
7 5 3 4 8 9 0 2 6 1
```

5.4 Apartado 4

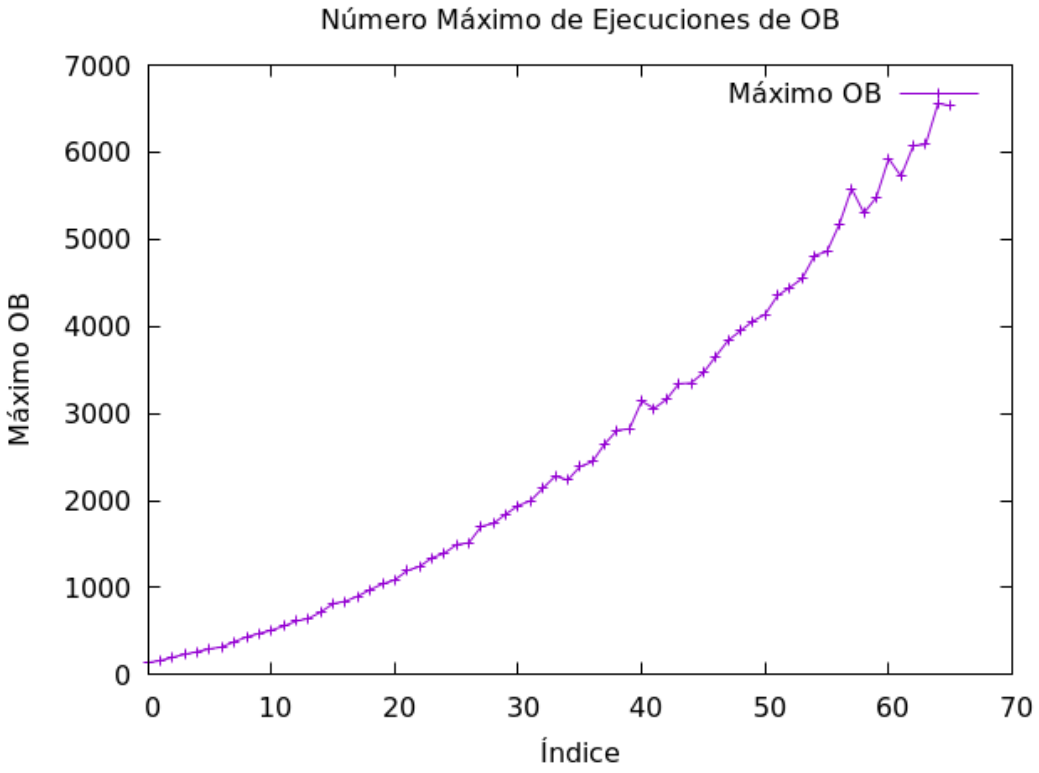
```
diego@diego-Victus-by-HP-Laptop-16-e0xxx:~/Documentos/universidad/anal/AALG-main$ ./exercise4 -size 8
Practice number 1, section 4
Done by: your names
Group: Your group
0       1       2       3       4       5       6       7
```

5.5 Apartado 5

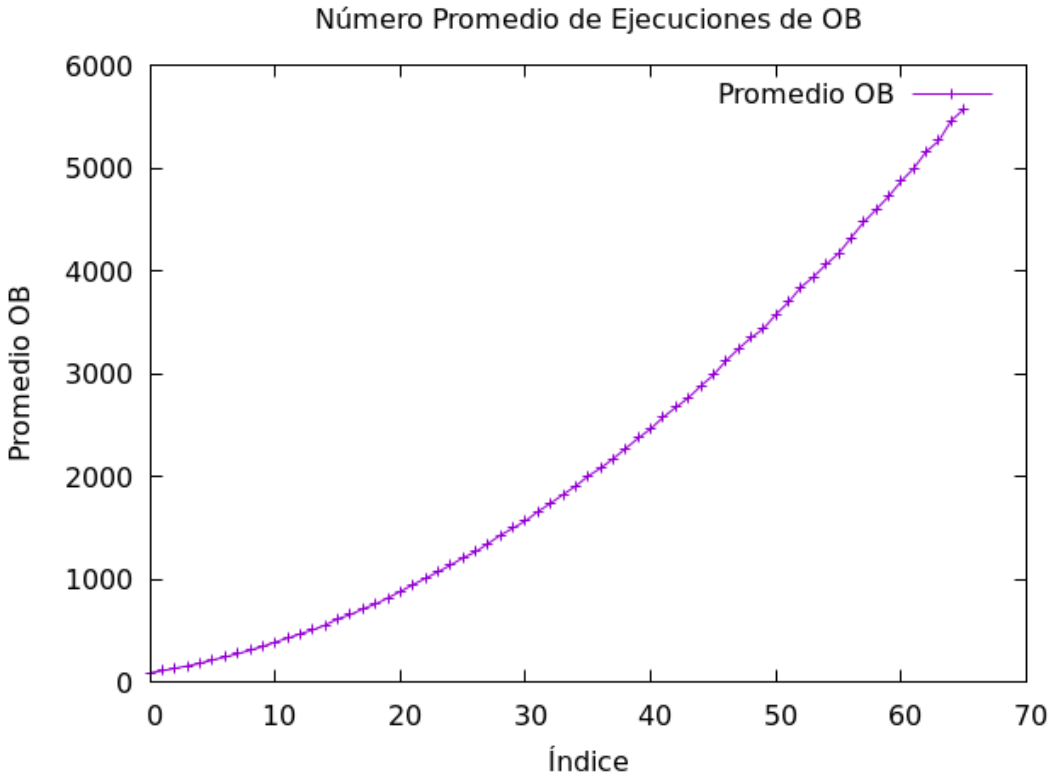
MÍNIMO



MÁXIMO:

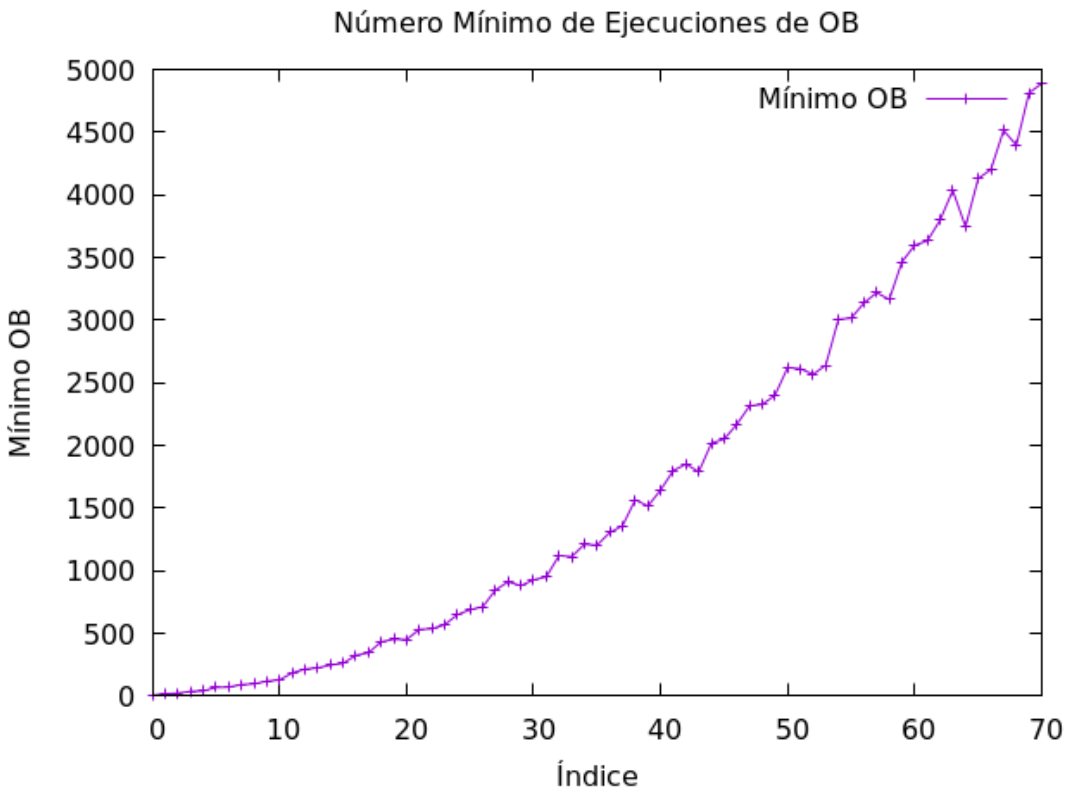


PROMEDIO:

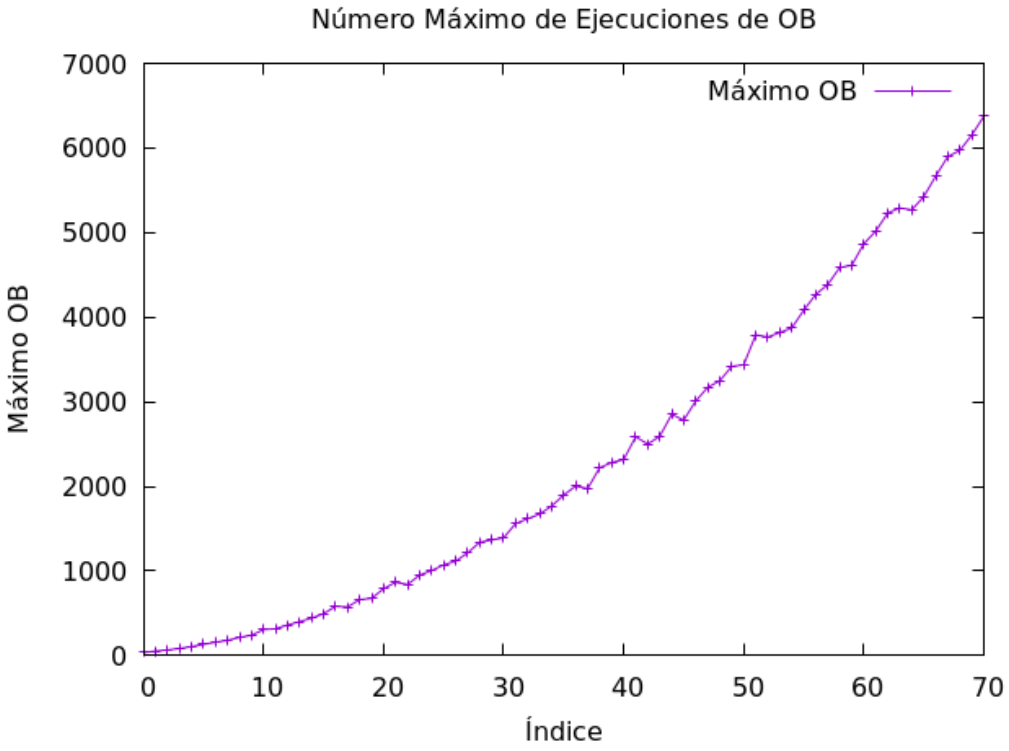


5.6 Apartado 6

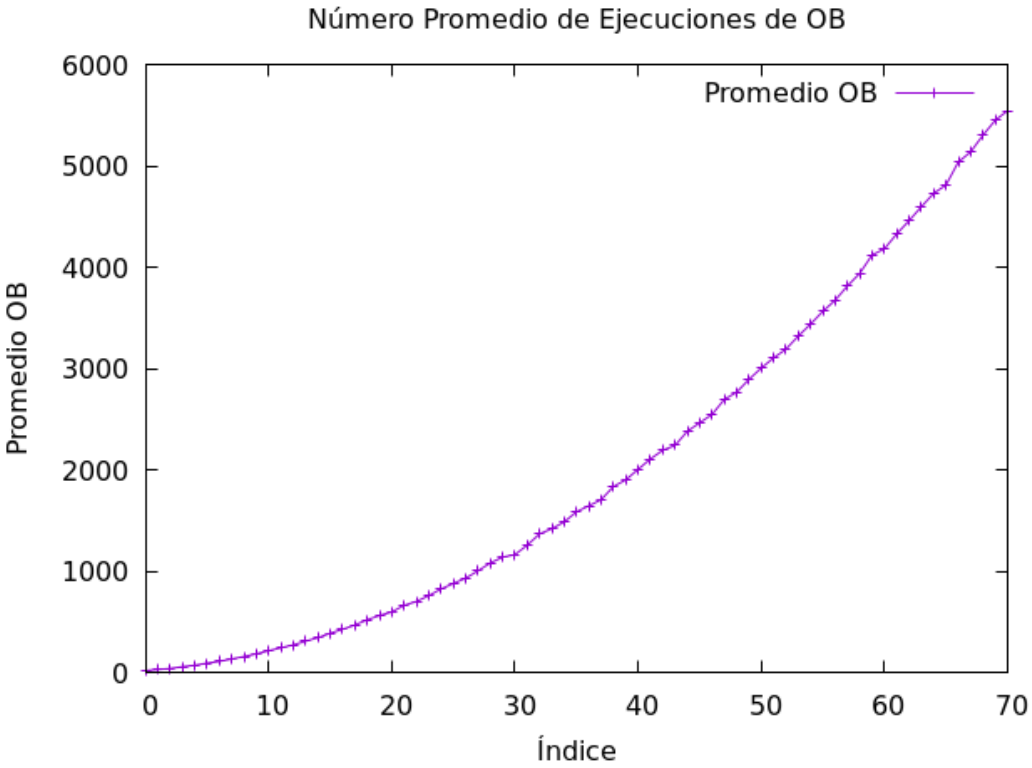
MÍNIMO:



MÁXIMO



PROMEDIO:



6. Respuesta a las preguntas teóricas.

6.1 Utilizamos rand () para generar un número aleatorio entre inf y sup. La lógica básica es calcular el rango (sup - inf) y luego sumar el valor mínimo inf para ajustar el número dentro del intervalo especificado.

6.2 Funciona comparando dos elementos de la tabla o array que le otorgas si el primer elemento es mayor (o menor, si se ordena en orden descendente) que el segundo, se intercambian, después de cada pasada por el arreglo, el elemento más grande (o más pequeño) se desplaza al final del array. Luego, se repite el proceso para los elementos restantes, excluyendo el último, que ya está en su posición correcta.

6.3 No actúa sobre el primer elemento porque el más grande ya está en su posición después de cada pasada.

6.4 Comparación e intercambio de elementos.

6.5 Peor caso (WBS): $O(n^2)$ para ambos algoritmos.

Mejor caso (BBS): O(n) para BubbleSortFlag, $O(n^2)$ para BubbleSort

6.5 *BubbleSortFlag* es más rápido en casos favorables (listas casi ordenadas) debido a su detección temprana de orden, pero similar en el peor caso.

7. Conclusiones finales.

La práctica entera se basa en la equiprobabilidad de los números aleatorios del primer bloque, aunque lo más importante bajo nuestro punto de vista es ser capaces de ver las diferencias no solo entre algoritmos, sino que también entre los diferentes casos que hay. Además, ha sido bastante útil para analizar costes de los algoritmos y darse cuenta de que no hay que usar la primear implementación que se te ocurra.