

Análisis de Algoritmos 2016/2017

Práctica 3

Simon valcarcel y Lucía Asencio, Grupo 6.

Código	Gráficas	Memoria	Total

1. Introducción.

En la práctica final de la asignatura continuamos con la evaluación del rendimiento de varios algoritmos, en esta ocasión sobre el TAD diccionario. Correspondiéndose con lo que hemos dado en clase los algoritmos son de búsqueda y no de ordenación.

2. Objetivos

2.1 Apartado 1

En el primer apartado se debe implementar el TAD diccionario, que consta de un array de claves, así como varios parámetros como si el array está ordenado, o el número de elementos en dicho array.

El ejercicio comienza implementando las primitivas para poder inicializar y eliminar el array, así como una función para introducir un elemento de manera ordenada o desordenada. Esta función se introduce en un bucle para generar la función de inserción masiva de datos en el diccionario.

Finalmente, implementamos tres funciones de búsqueda: Búsqueda lineal, búsqueda lineal auto-organizada, que cuando encuentra un elemento, lo traslada una posición menos en el array, de manera que búsquedas muy frecuentes tengas un coste $O(1)$. También implementamos búsqueda binaria, un elegante método parecido a métodos de divide y vencerás que en cada iteración reduce el tamaño sobre el conjunto en el que se busca por la mitad.

2.2 Apartado 2

Como en las dos prácticas anteriores, hay que conseguir datos del rendimiento de los distintos métodos de ordenación. Para ello, implementamos una función que genere los OB max, min y medios, así como el tiempo de búsqueda para un método de búsqueda sobre un array en concreto.

La segunda función implementada ejecuta el método de ordenación sobre arrays de tamaños incrementalmente mayores. Estos datos, guardados en un fichero con un formato adecuado, se pueden usar para obtener gráficas de rendimiento.

3. Herramientas y metodología

3.1 Apartado 1

Comenzamos diseñando el TAD en gedit, compilándolo paulatinamente para comprobar el funcionamiento adecuado de cada función, ejecutándose el archivo resultante con valgrind memcheck.

Tuvimos un ligero problema a la hora de implementar la inserción ordenada por un error trivial pero muy difícil de encontrar.

También experimentamos algunos problemas de memoria en la búsqueda lineal auto organizada debido a un swap mal implementado. Con el desglose de los errores de Valgrind lo resolvimos sin problemas.

3.2 Apartado 2

Una vez implementado el TAD diccionario, y usando las funciones de generar tiempo de ordenación de los ejercicios pasados como plantillas, no supuso ningún problema implementar las modificaciones requeridas por el ejercicio.

Las gráficas, una vez obtenidos los datos, las realizamos con GNUplot.

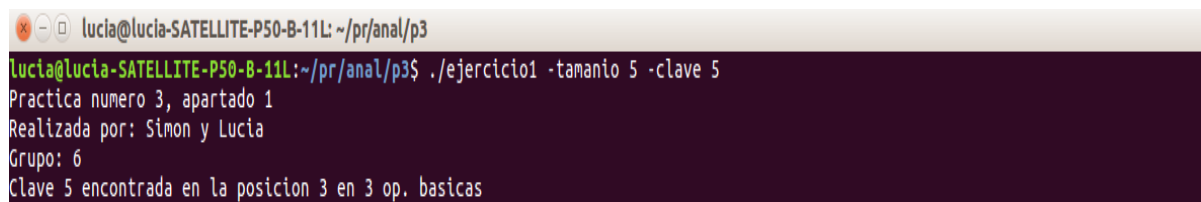
4. Código fuente

Se encuentra al final de la memoria.

5. Resultados, Gráficas

5.1 Apartado 1

El programa del ejercicio 1 ejecuta correctamente:

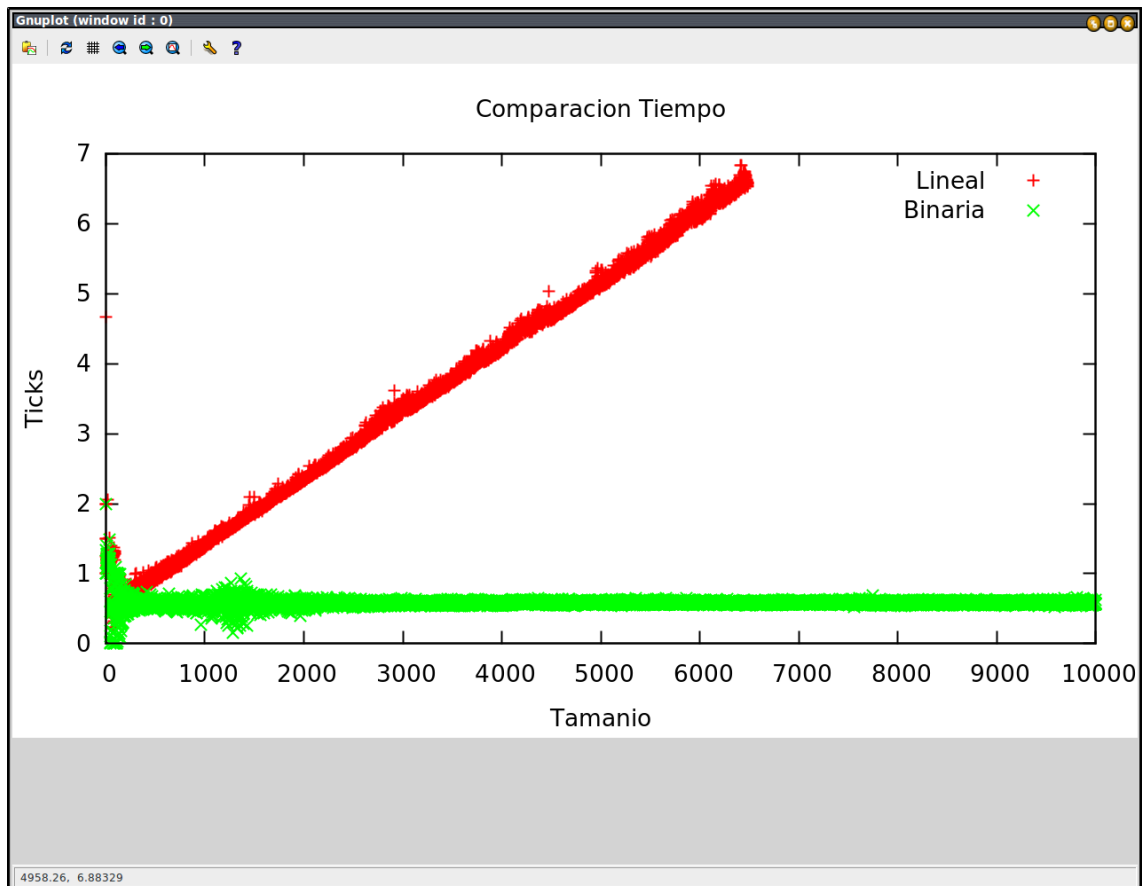
A terminal window with a dark background and light text. The title bar shows 'lucia@lucia-SATELLITE-P50-B-11L: ~/pr/anal/p3'. The terminal content shows a command being executed and its output.

```
lucia@lucia-SATELLITE-P50-B-11L: ~/pr/anal/p3
lucia@lucia-SATELLITE-P50-B-11L:~/pr/anal/p3$ ./ejercicio1 -tamaño 5 -clave 5
Practica numero 3, apartado 1
Realizada por: Simon y Lucia
Grupo: 6
Clave 5 encontrada en la posicion 3 en 3 op. basicas
```

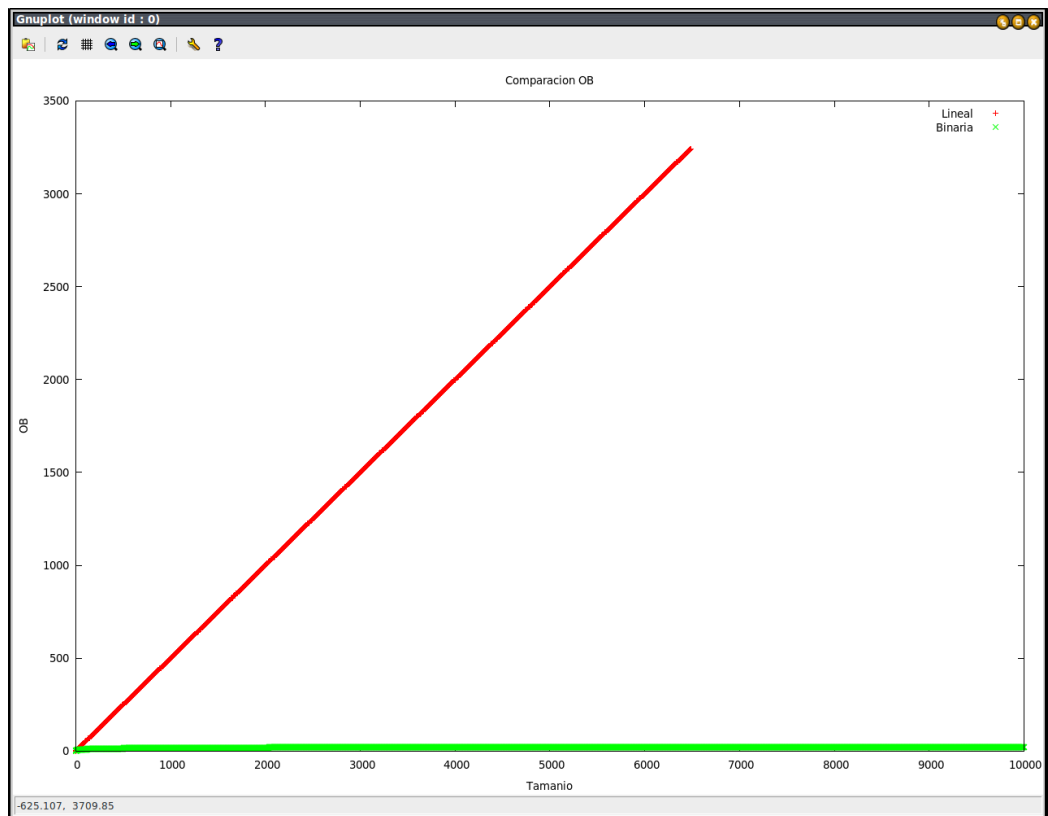
5.2 Apartado 2

Resultados del apartado 2.

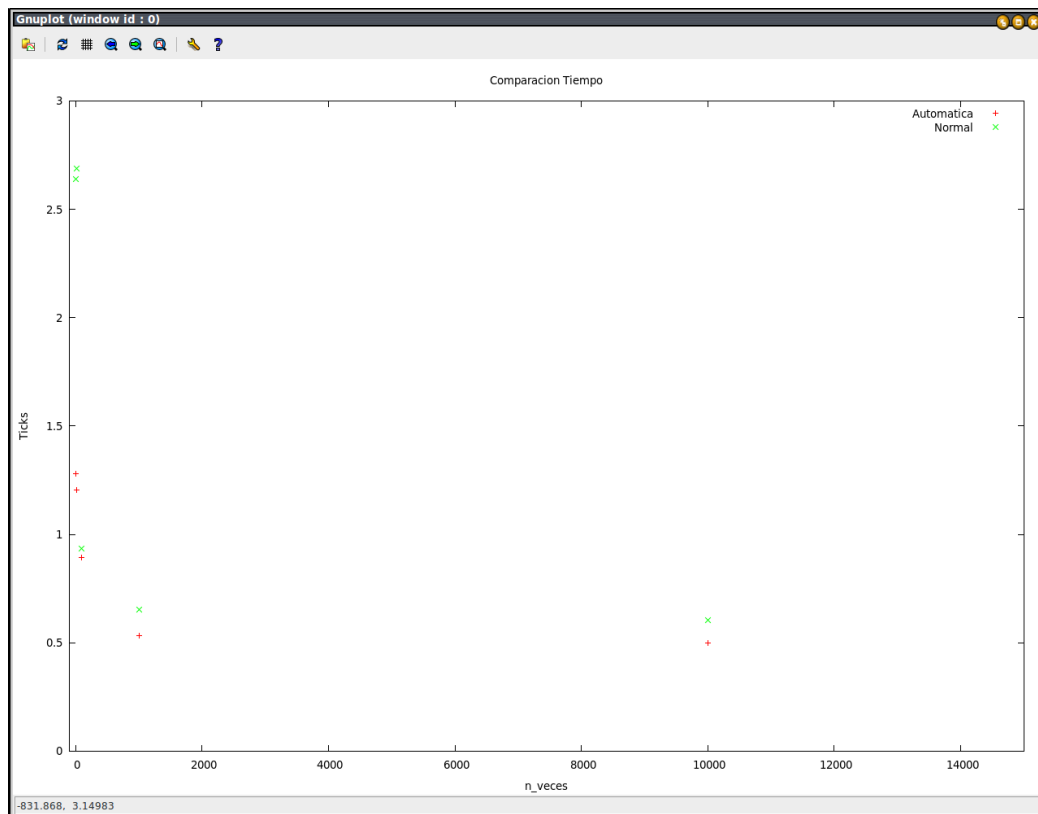
Primero realizamos las gráficas que pide **el enunciado**:



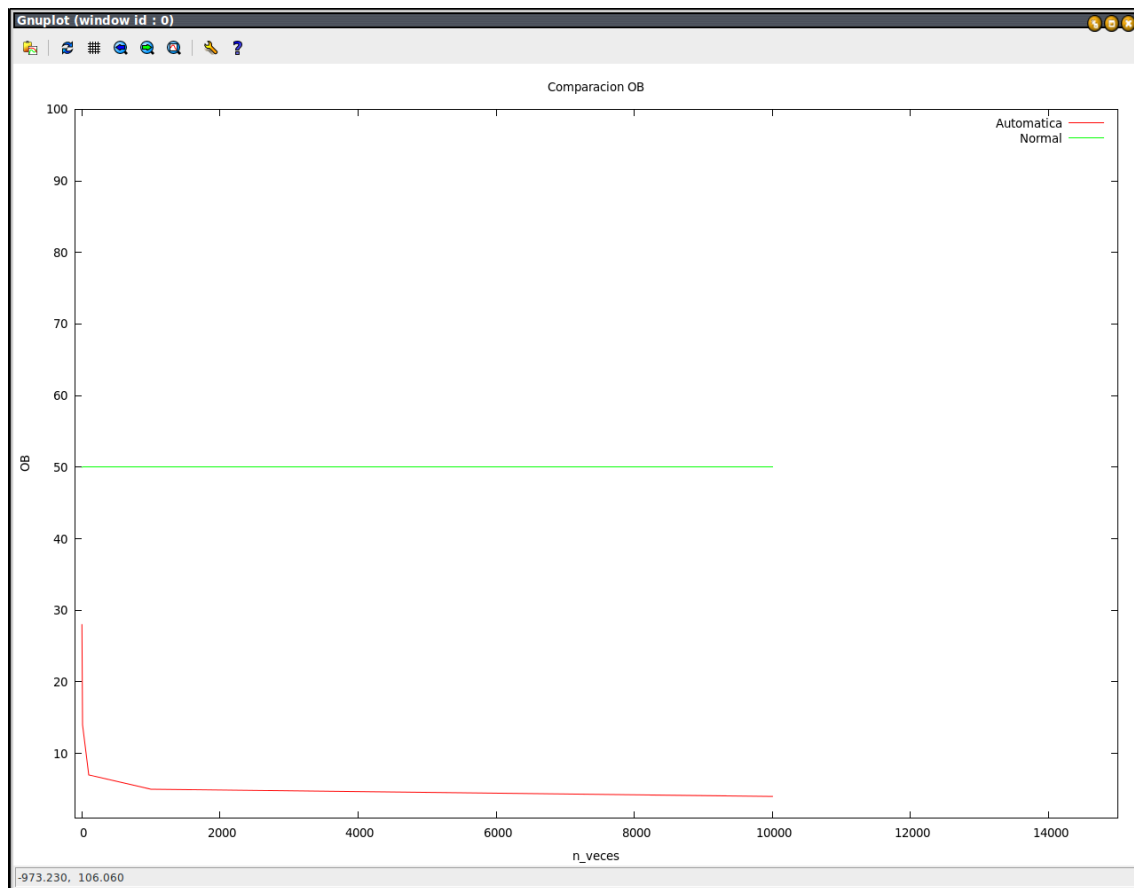
Podemos ver que Mientras que la lineal se dispara, la Binaria se mantiene con un incremento tan pequeño que no es notable



El patrón se mantiene, y mientras que la búsqueda lineal crece, valga la redundancia linealmente. La gráfica se parte porque tiempos de Binaria se obtienen en poco tiempo, pero para la lineal, el coste se dispara.

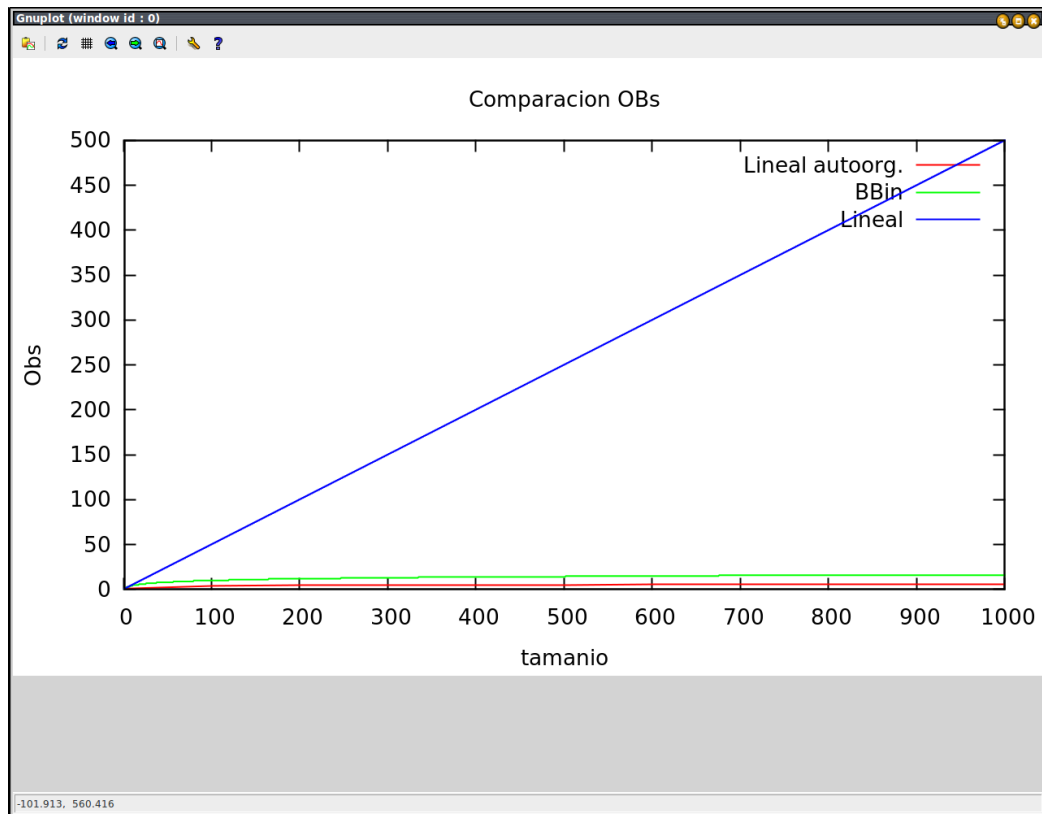


Podemos observar que la automática siempre se coloca por debajo de la automática, lo que confirma nuestras sospechas teóricas.

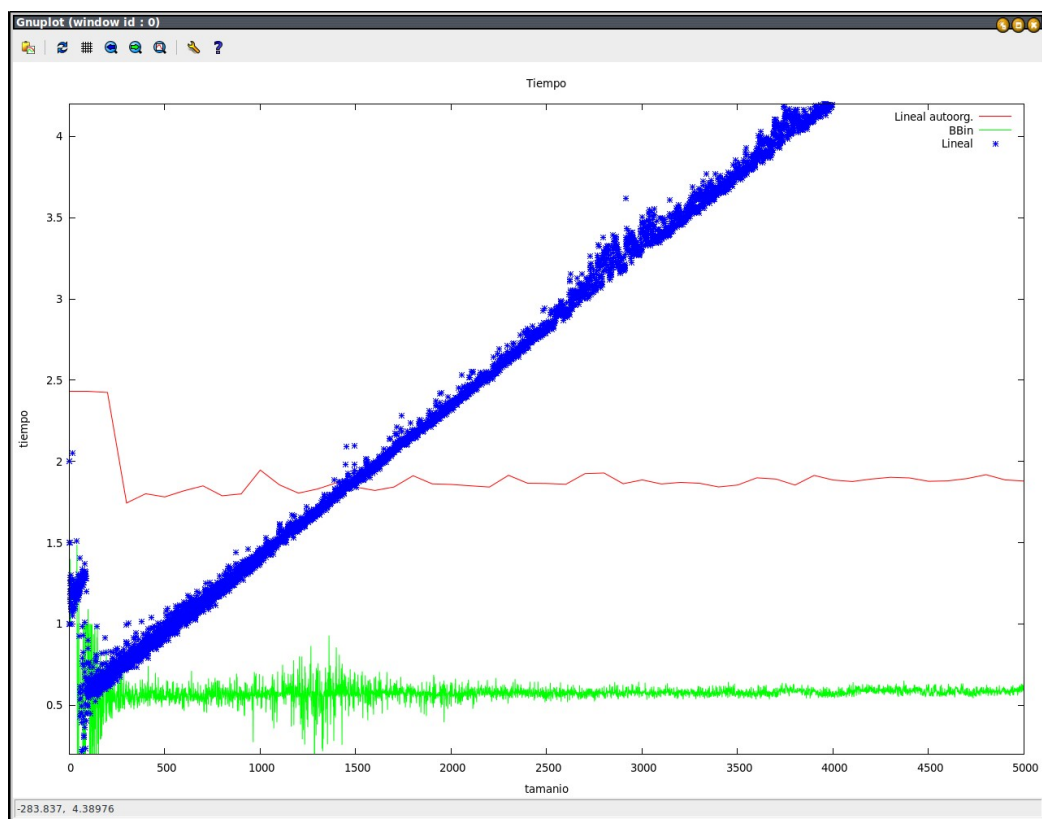


Lo que no se veía claro en el gráfico anterior sí se ve claro ahora. Mientras que el ob medio de la normal se mantiene en 50, la organizada desciende rápidamente a 5.

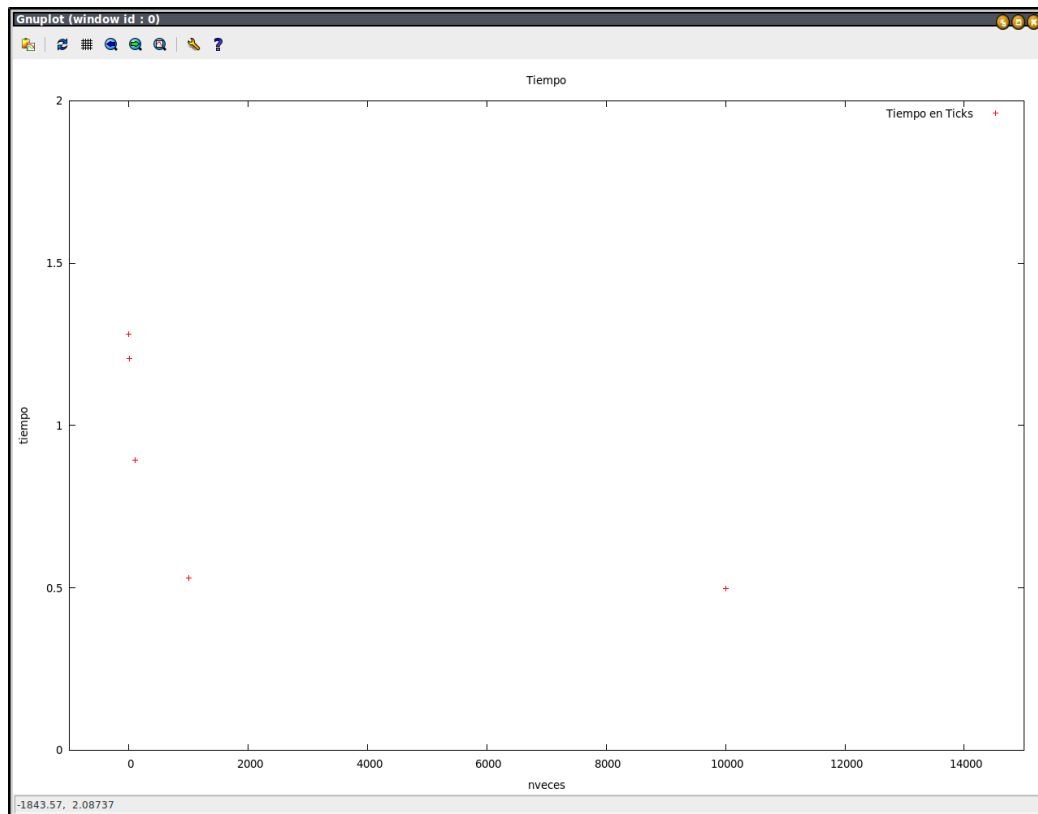
Ahora realizamos las gráficas que pide **la memoria**



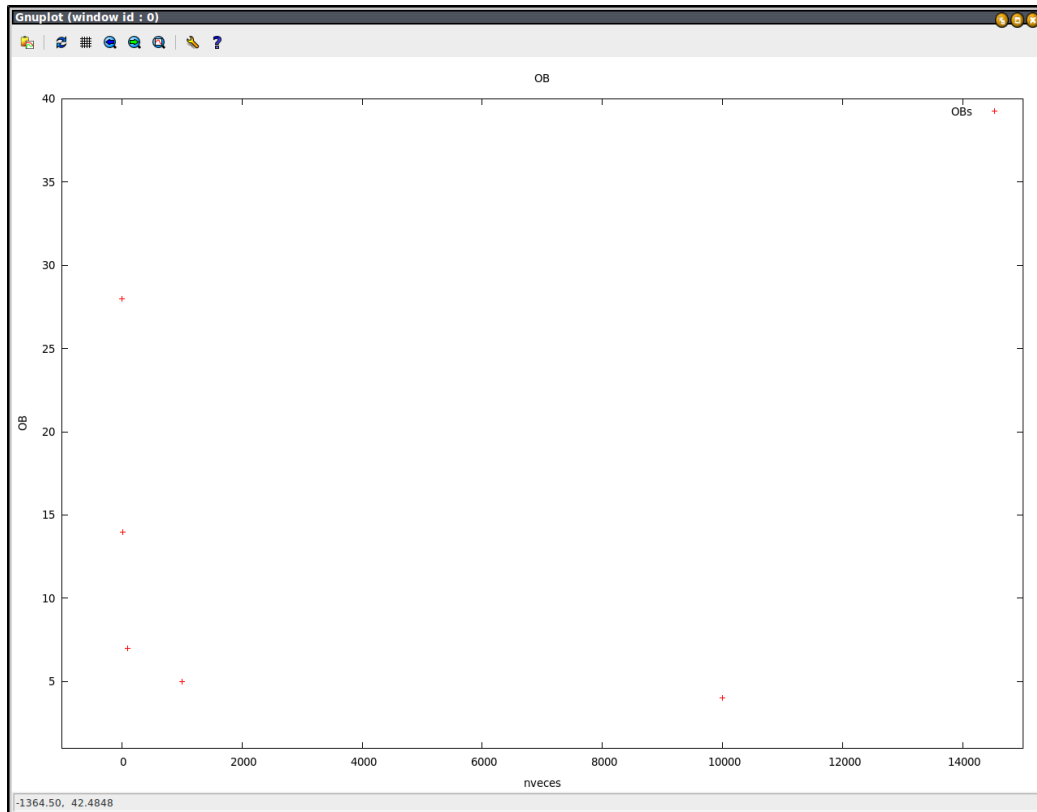
Como podemos ver, la binaria y la auto organizada se mantienen cerca de valores mínimos, mientras que la lineal normal progresa con crecimiento lineal.



La gráfica de los tiempos presenta un comportamiento análogo a la anterior, con la salvedad de que tuvimos que ampliar el rango a 5000 para que se vea que blin tarda más que el resto.



Como se puede observar, los ticks disminuyen gravemente a medida que el número de n_veces aumenta.



El comportamiento de los OBs refleja el de los tiempos, disminuyendo notablemente a medida que las repeticiones aumentan.

5. Respuesta a las preguntas teóricas.

5.1 Pregunta 1

La OB en las tres funciones es la CDC o comparación de claves.

5.2 Pregunta 2

Para **busqueda lineal**, el mejor caso(el elemento está al principio) es $O(1)$. EL peor caso(el elemento está al final) es $n-1$, es decir $O(n)$

Para **búsqueda binaria**, el mejor caso (el elemento a buscar está en el medio) vuelve a ser $O(1)$ y el peor caso es $O(\log n)$

5.3 Pregunta 3

La posición de los elementos más buscados va haciéndose progresivamente más cercana al inicio del array, hasta que después de suficientes búsquedas, el elemento más buscado esté en la primera posición.

5.4 Pregunta 4

A medida que se realizan más búsquedas, puesto que estas no responden a una distribución uniforme y se le da prioridad a las más buscadas, el coste acaba siendo de $O(1)$

5.5 Pregunta 5

Bbin realiza correctamente sus búsquedas porque el array ya está ordenado. Este algoritmo de búsqueda se basa en lo siguiente: En cada búsqueda, tiene una cota inferior y superior sobre las posiciones del array entre las que va a buscar. Comienza haciendo la media entre ambas y establece si el elemento buscado se encuentra en esta posición media. Si esto no es el caso, establece si el elemento medio es menor o mayor que el buscado. Se pueden dar dos casos:

- El elemento medio es menor que el elemento buscado. Entonces el elemento medio actúa como nueva cota inferior y la cota superior se mantiene.

- El elemento medio es mayor que el elemento medio. Entonces el elemento medio actúa como la nueva cota superior y la cota inferior se mantiene.

El programa termina su ejecución cuando encuentra el elemento buscado o cuando la cota inferior es igual a la mayor, indicando que el elemento no se ha encontrado.

Los dos condicionales previos sólo tiene sentido si trabajamos sobre una lista ordenada, de lo contrario, no tiene sentido “dividir” el array, con la garantía de que el elemento buscado está en la parte de array que nos quedamos.

6. Conclusiones finales.

Concluimos la última práctica de Análisis de Algoritmos con una comprensión básica pero completa del funcionamiento de algoritmos de búsqueda y de ordenación, además de una manera empírica de determinar el rendimiento de estos. Estamos seguros de que todavía queda mucho por aprender hasta poder entender completamente el mundo de los algoritmos, pero estamos convencidos de que estas prácticas han sido un paso importante en la dirección correcta.

Apéndice: El código

Ejercicio 1

```
PDICC ini_diccionario (int tamaño, char orden){
    if(tamaño<0) return NULL;
    int i=0;
    PDICC d;
    d= (PDICC) malloc(sizeof(DICC));
    d->tamaño=tamaño;
    d->orden=orden;
    d->n_datos=0;
    d->orden= orden;
    d->tabla= (int*) malloc(tamaño*sizeof(int));
    if(! d->tabla) return NULL;
    for(i=0; i>tamaño; i++)
        d->tabla[i]=0;

    return d;
}
```

```
void libera_diccionario(PDICC pdicc){
    if(!pdicc)
        return;
    if(pdicc->tabla)
        free(pdicc->tabla);
    free(pdicc);
}
```

```
int inserta_diccionario(PDICC pdicc, int clave){
    int i=0, ob=0;
    if(!pdicc) return ERR;
```

```

if(pdicc->orden== NO_ORDENADO){
    pdicc->tabla[++(pdicc->n_datos)-1]=clave;
    return 0;
}
else if(pdicc->orden==ORDENADO){

    pdicc->tabla[pdicc->n_datos] = clave;
    pdicc->n_datos ++;
    i = pdicc->n_datos-2;
    while(i >= 0 && pdicc->tabla[i]> clave){
        pdicc->tabla[i+1] = pdicc->tabla[i];
        i--;
        ob++;
    }
    pdicc->tabla[i+1] = clave;
    return ob+1;
}
return ERR;
/*
    (pdicc->n_datos)++;
    pdicc->tabla[pdicc->n_datos-1]=clave;
    for(i=pdicc->n_datos-2; i>=0; i--){
        if(pdicc->tabla[i]<pdicc->tabla[i+1])
            return 1;
        if(i > 0)
            swap(pdicc->tabla, i, i-1);
    }
}

```

```

        if(i!=-1)
            return 0;
        else
            return 1;*/
    }

```

```

void swap(int * arr, int p1, int p2){
    if(!arr) return;
    int i;
    i= arr[p1];
    arr[p1]= arr[p2];
    arr[p2]= i;
    return;
}

```

```

int insercion_masiva_diccionario (PDICC pdicc,int *claves, int n_claves){
    int i, ob=0, obbuff;
    if(!claves || !pdicc || n_claves<1) return 0;
    if(n_claves+pdicc->n_datos > pdicc->tamano)
        return ERR;

    for(i=0; i<n_claves; i++){
        obbuff = inserta_diccionario(pdicc, claves[i]);
        if(obbuff == ERR)
            return ERR;
        ob += obbuff;
    }
}

```

```

        return ob;
    }

void print_tabla(PDICC pdicc){

    int i;
    if(!pdicc) return;
    for(i=0; i<pdicc->n_datos; i++)
        printf("%d ", pdicc->tabla[i]);
        printf("\n");
    }

int busca_diccionario(PDICC pdicc, int clave, int *ppos, pfunc_busqueda metodo){
    if(!pdicc || !metodo) return ERR;

    return (* metodo)(pdicc->tabla, 0, pdicc->n_datos - 1, clave, ppos);
}

void imprime_diccionario(PDICC pdicc){
    if(pdicc) return;
    printf("Tamaño: %d\n", pdicc->tamano);
    printf("Número de datos: %d\n", pdicc->n_datos);
    printf("Ordenado: %c\n", pdicc->orden);
    print_tabla(pdicc);

}

/* Funciones de busqueda del TAD Diccionario */
int bbin(int *tabla,int P,int U,int clave,int *ppos){
    int ob = 1;
    int medio;

```



```

    if(!tabla || P>U || U<0) return ERR;

    *ppos = NO_ENCONTRADO;
    medio = (P+U)/2;
    if (clave == tabla[medio])
        *ppos = medio+1;
    else if(clave < tabla[medio] )
        ob = 2 + bbin(tabla, P, medio-1, clave, ppos);
    else
        ob = 2 + bbin(tabla, medio+1, U, clave, ppos);
    return ob;

}

```

```

int blin(int *tabla,int P,int U,int clave,int *ppos){
    int ob = 0;
    int i;
    if(!tabla || P>U || U<0) return ERR;

    *ppos = NO_ENCONTRADO;
    for(i = P; (i <= U) && (clave!=tabla[i]); ob ++, i++);
    *ppos = i+1;
    ob++;
    return ob;
}

```

/*Esta función no DEVUELVE en ppos la POSICION DE LA CLAVE en la tabla inicial,

sino UNA VEZ SWAPEADA*/

```

int blin_auto(int *tabla,int P,int U,int clave,int *ppos){

```

```

int ob = 1;

int i;

if(!tabla || P>U || U<0) return ERR;

*ppos = NO_ENCONTRADO;

for(i = P; i <= U; ob ++, i++){
    if(clave == tabla[i]){
        if(i>0){
            swap(tabla, i, i-1);

            *ppos = i-1;

            break;

        }else{
            *ppos=i;

            break;}

    }

}

return ob;

}

```

Ejercicio 2:

```

/*****/

/* Funcion: tiempo_medio_ordenacion Fecha: */
/* Parametros: metodo de busqueda a utilizar, funci*/
/* generadora de claves, orden y tamaño del diccio */
/* nario a utilizar, numero de claves a buscar nve */
/* ces, ptiempo en el que gardar resultados */
/*****/

short tiempo_medio_busqueda(pfunc_busqueda metodo, pfunc_generador_claves
generador,

                        int orden, int tamano, int n_claves, int n_veces,
                        PTIEMPO ptiempo){

    PDICC d = NULL;
    int* perm, claves[n_veces*n_claves];
    int i, pos, obBuff;
    long long obMed = 0;
    float t1, t2, tResta, tMed = 0;

    if(!metodo || !generador || (orden != 1 && orden != 0) || n_claves<1
    || tamano<1 || n_veces<1 || !ptiempo) return ERR;

    /*Inicializamos valores conocidos de PTIEMPO*/
    ptiempo->n_perms=n_claves;
    ptiempo->tamano= tamano;
    ptiempo->n_veces = n_veces;
    ptiempo->min_ob= INT_MAX;
    ptiempo->max_ob= 0;

```

```

/*Creamos dicc,
creamos permutacion y la insertamos,
creamos tabla de claves a buscar*/
d = ini_diccionario(tamano, orden);
if(d == NULL) return ERR;
perm = genera_perm(n_claves);
if(perm == NULL){
    libera_diccionario(d);
    return ERR;
}
if(insertion_masiva_diccionario(d, perm, n_claves) == ERR){
    libera_diccionario(d);
    free(perm);
    return ERR;
}
generador(claves, n_claves*n_veces, n_claves);

/*Bucle de llamadas a metodo*/
for(i = 0; i < n_claves*n_veces; i++){
    /*LLamada a metodo, midiendo el tiempo de ejecucion*/
    t1 = clock();
    obBuff = busca_diccionario(d, claves[i], &pos, metodo);
    if(obBuff == ERR){
        libera_diccionario(d);
        free(perm);
        return ERR;
    }
    t2 = clock();

    /*Actualizacion de los valores de PTIEMPO*/
    if(obBuff<ptiempo->min_ob){

```

```

        ptiempo->min_ob=obBuff;
    }
    if(ptiempo->max_ob<obBuff){
        ptiempo->max_ob=obBuff;
    }
    obMed+= obBuff;
    tResta = (t2 - t1);
    tMed+=tResta;
}

ptiempo->tiempo=tMed/(n_claves * n_veces);
ptiempo->medio_ob=obMed/(n_claves * n_veces);

/*Liberamos recursos*/
libera_diccionario(d);
free(perm);
return OK;
}

```

```

/*****/

/* Funcion: genera_tiempos_busqueda Fecha: */
/*Parametros: metodo de busqueda, funcion generado */
/*ra de claves a usar, orden del diccionario a cre */
/*ar, fichero en el que escribir resultados, num_m */
/*in y num_max los tamaños entre los que varia el */
/*diccionario, incremento entre sus tamaños. */
/* numero de veces que se buscara cada clave */

```

```

/*****/

short genera_tiempos_busqueda(pfunc_busqueda metodo, pfunc_generador_claves
generador,

    int orden, char* fichero, int num_min, int num_max,

    int incr, int n_veces){

    int iter=0, i, j, tamano;

    iter=((num_max-num_min)/incr)+1;
    PTIEMPO * pptiempo= (PTIEMPO *)malloc(sizeof(PTIEMPO)*iter);
    if(!pptiempo) return ERR;

    if(!fichero || !metodo || !generador || (orden != 1 && orden != 0) || num_min<1
    || num_min>num_max || incr<1 || n_veces<1)
    return ERR;

    /*Generamos array de PTIEMPO*/
    for(i=0; i<iter; i++){
        pptiempo[i]= (PTIEMPO) malloc(sizeof(TIEMPO));
        if(!pptiempo[i]){
            for(j = 0; j<i; j++){
                free(pptiempo[j]);
            }
            free(pptiempo);
            return ERR;
        }
    }

    /*Hacemos tantas llamadas a tiempo_medio_busqueda como iteraciones,
    guardamos los resultados*/
    for(i=0, tamano=num_min; i<iter; i++, tamano+=incr){

```

```
        tiempo_medio_busqueda( metodo, generador, orden, tamano, tamano,
                                n_veces, pptiempo[i]);
        guarda_tabla_tiempos(fichero, pptiempo[i], 1);
    }

    /*Liberamos recursos*/
    for(i = 0; i<iter; i++){
        if(pptiempo[i])
            free(pptiempo[i]);
    }
    free(pptiempo);
    return OK;
}
```