Análisis de Algoritmos 2018/2019

Práctica 3

Andrés Mena Godino y Juan Moreno Díez,  Grupo 1272.

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

**1. Introducción.**

En esta práctica implementaremos el TAD diccionario mediante el uso del módulo **búsqueda.c** y las funciones definidas en su respectivo archivo de cabecera. Utilizaremos este TAD junto a la implementación de dos funciones nuevas en **tiempos.c** y las rutinas de búsqueda (búsqueda lineal y búsqueda bibaria). Con estas incorporaciones mediremos número máximo, mínimo y medio de operaciones básicas, tiempo medio y operaciones básicas medias de la búsqueda binaria y búsqueda lineal.

**2. Objetivos**

2.1 Apartado 1

El objetivo principal es crear un **TAD diccionario** que tendremos definido en el archivo de cabecera **busqueda.h** que nos proporciona el enunciado. Tendremos una estructura llamada DICC que se utilizará para implementar el diccionario. Luego desarrollamos en **busqueda.c** las rutinas que tenemos definidas. Todas las funciones devolverán el número de OBs o ERR en caso de error, excepto **ini\_diccionario** que devolverá NULL si existe error o el diccionario construido si todo se ha realizado correctamente. El programa ejercicio1.c nos sirve para comprobar el funcionamiento de la búsqueda binaria (que se aplica sobre tablas ordenadas), y la búsqueda lineal (que se aplica sobre arrays desordenados).

2.2 Apartado 2

En este apartado implementaremos dos funciones para realizar medidas de rendimiento de las funciones de búsqueda que hemos desarrollado previamente. Éstas serán: **short tiempo\_medio\_busqueda(pfunc\_busqueda metodo, pfunc\_generador\_claves generador, char orden, int N, int n\_veces, PTIEMPO ptiempo)** y **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).** La primera rutina devuelve ERR en caso de error y OK en el caso de que las búsquedas se realicen correctamente. La segunda función sirve para automatizar la toma de tiempos. Esta, llama a la función **tiempo\_medio\_busqueda** con un tamaño de diccionario desde num\_min hasta num\_max, usando incrementos de tamaño incr. La rutina devolverá el valor ERR en caso de error y OK en caso contrario. Por último, guardará los resultados en el fichero mediante la función **short guarda\_tabla\_tiempos.** Habrá que comprobar el número máximo, mínimo y medio de OBs de la búsqueda binaria y de la búsqueda lineal y el tiempo medio y operaciones básicas medias de las búsquedas binaria y lineal.

**3. Herramientas y metodología**

Aquí ponéis qué entorno de desarrollo (Windows, Linux, MacOS) y herramientas habéis utilizado (Netbeans, Eclipse, gcc, Valgrind, Gnuplot, Sort, uniq, etc) y qué metodologías de desarrollo y soluciones al problema planteado habéis empleado en cada apartado. Así como las pruebas que habéis realizado a los programas desarrollados.

3.1 Apartado 1

Hemos utilizado Linux como entorno de desarrollo, más en concreto Ubuntu ya que facilita la organización de código y la programación. Para la implementación de código hemos utilizado Atom. Utilizamos gcc para la compilación y Valgrind para comprobar las pérdidas de memoria. Por último, hemos usado gnuplot para representar las gráficas necesarias.

3.2 Apartado 2

Hemos utilizado Linux como entorno de desarrollo, más en concreto Ubuntu ya que facilita la organización de código y la programación. Para la implementación de código hemos utilizado Atom. Utilizamos gcc para la compilación y Valgrind para comprobar las pérdidas de memoria. Por último, hemos usado gnuplot para representar las gráficas necesarias.

**4. Código fuente**

Aquí ponéis el código fuente **exclusivamente de las rutinas que habéis desarrollado vosotros** en cada apartado.

4.1 Apartado 1

PDICC ini\_diccionario (int tamanio, char orden){

PDICC diccionario = (PDICC) malloc (sizeof(DICC));

if (diccionario == NULL)

return NULL;

diccionario->tamanio = tamanio;

diccionario->orden = orden;

diccionario->n\_datos = 0;

diccionario->tabla = (int \*)malloc(tamanio\*sizeof(diccionario->tabla[0]));

if (diccionario->tabla == NULL){

free (diccionario);

return NULL;

}

return diccionario;

}

void libera\_diccionario(PDICC pdicc)

{

free(pdicc->tabla);

free(pdicc);

}

int inserta\_diccionario(PDICC pdicc, int clave)

{

int obs=1;

if (pdicc->tamanio <= pdicc->n\_datos)

return ERR;

pdicc->tabla[pdicc->n\_datos] = clave;

pdicc->n\_datos += 1;

if (pdicc -> orden == ORDENADO){

int j = pdicc->n\_datos - 2;

while (j >= 0 && pdicc->tabla[j]>clave){

obs+=1;

pdicc->tabla[j+1]=pdicc->tabla[j];

j--;

}

obs+=1;

pdicc->tabla[j+1]=clave;

}

return obs;

}

int insercion\_masiva\_diccionario (PDICC pdicc,int \*claves, int n\_claves){

int i, obs\_aux, obs=0;

for (i=0; i < n\_claves; i++){

if ((obs\_aux=inserta\_diccionario(pdicc, claves[i]))==ERR)

return ERR;

obs+=obs\_aux;

}

return obs;

}

int busca\_diccionario(PDICC pdicc, int clave, int \*ppos, pfunc\_busqueda metodo)

{

return metodo (pdicc->tabla, 0, pdicc->n\_datos, clave, ppos);

}

int bbin(int \*tabla,int p,int u,int clave,int \*ppos){

int m, obs=0;

if (p>u)

return ERR;

while (p <= u){

m = (p+u)/2;

obs+=1;

if (tabla[m] == clave){

\*ppos = m;

return obs;

}

else{

if (tabla[m] > clave)

u = m - 1;

else

p = m + 1;

}

}

\*ppos=NO\_ENCONTRADO;

return obs;

}

int blin(int \*tabla, int p, int u, int clave, int \*ppos){

int i, obs=0;

for(i=p; i < u; i++){

obs+=1;

if(clave == tabla[i]){

\*ppos = i;

return obs;

}

}

\*ppos=NO\_ENCONTRADO;

return obs;

}

4.2 Apartado 2

short tiempo\_medio\_busqueda(pfunc\_busqueda metodo, pfunc\_generador\_claves generador, int orden, int N, int n\_veces, PTIEMPO ptiempo){

int i, ppos, obt, \*perm, \*claves;

double start, end;

PDICC pdiccionario = ini\_diccionario(N, orden);

if (pdiccionario == NULL)

return ERR;

perm = genera\_perm(N);

if (perm == NULL) {

libera\_diccionario(pdiccionario);

return ERR;

}

if (insercion\_masiva\_diccionario(pdiccionario, perm, N) == ERR) {

libera\_diccionario(pdiccionario);

free(perm);

return ERR;

}

claves = (int\*)malloc(n\_veces\*N\*sizeof(int));

if (claves == NULL){

libera\_diccionario(pdiccionario);

free(perm);

return ERR;

}

generador(claves, N\*n\_veces, N);

ptiempo->medio\_ob = 0;

ptiempo->max\_ob = 0;

ptiempo->min\_ob = INT\_MAX;

ptiempo->n\_elems = n\_veces\*N;

start=clock();

for (i=0; i<N\*n\_veces; i++){

if ((obt=metodo(pdiccionario->tabla, 0, N-1, claves[i], &ppos))==ERR){

libera\_diccionario (pdiccionario);

free (claves);

free (perm);

return ERR;

}

ptiempo->medio\_ob+=(double)obt;

if (obt>ptiempo->max\_ob)

ptiempo->max\_ob = obt;

if (obt<ptiempo->min\_ob)

ptiempo->min\_ob = obt;

}

end=clock();

ptiempo->medio\_ob/=ptiempo->n\_elems;

ptiempo->tiempo = (double)(end-start)/ptiempo->n\_elems/CLOCKS\_PER\_SEC;

ptiempo->N = N;

libera\_diccionario (pdiccionario);

free (claves);

free (perm);

return OK;

}

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){

PTIEMPO ptiempo;

int j, i;

int n=(num\_max-num\_min)/incr +1;

ptiempo=(PTIEMPO) malloc (n \* sizeof(ptiempo[0]));

if (ptiempo==NULL)

return ERR;

for (j=num\_min, i=0 ; j<=num\_max ; j+=incr, i++){

if (tiempo\_medio\_busqueda (metodo, generador, orden, j, n\_veces, &ptiempo[i])==ERR){

free (ptiempo);

return ERR;

}

}

if (guarda\_tabla\_tiempos(fichero, ptiempo, i)==ERR){

free (ptiempo);

return ERR;

}

free (ptiempo);

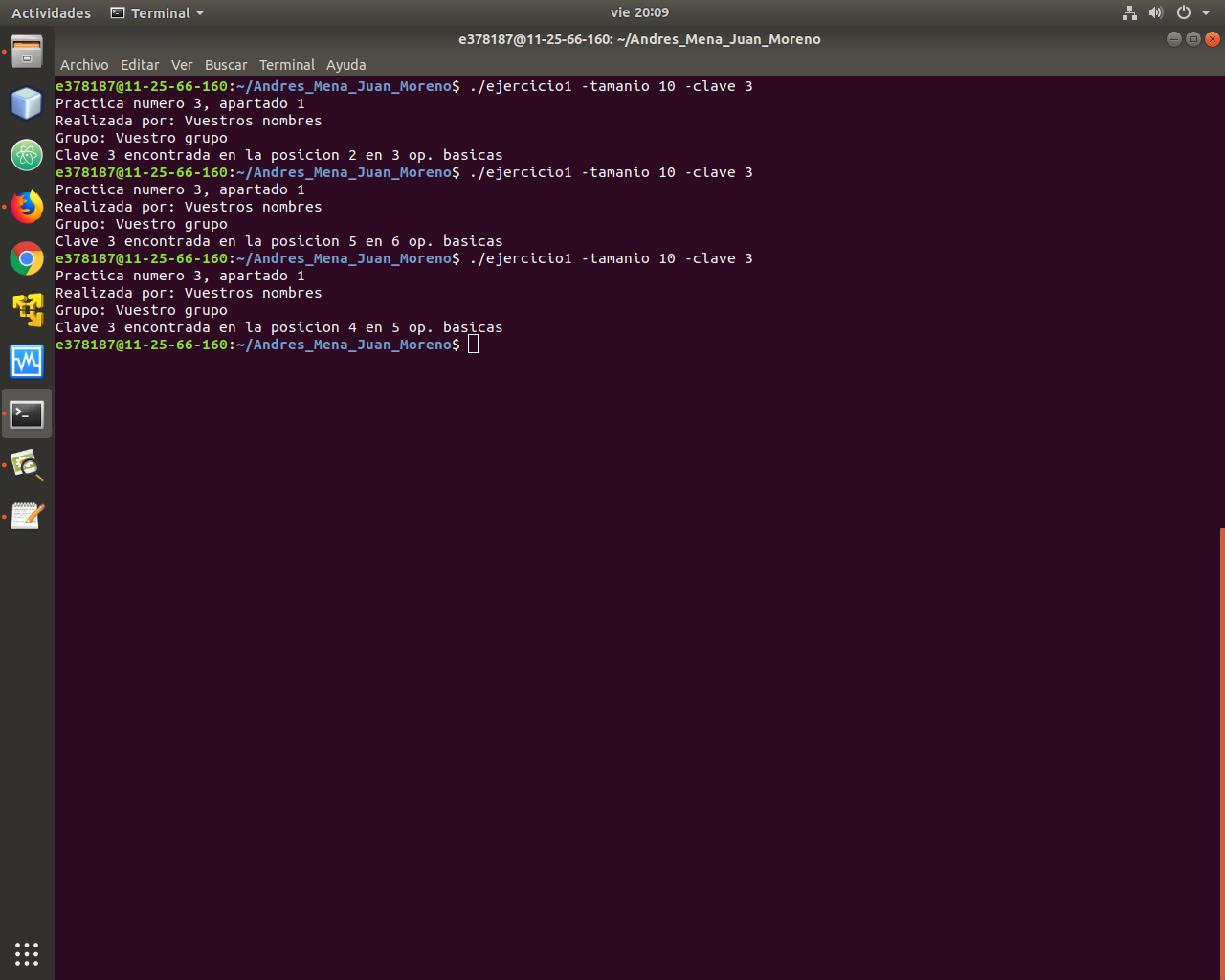
return OK;

}

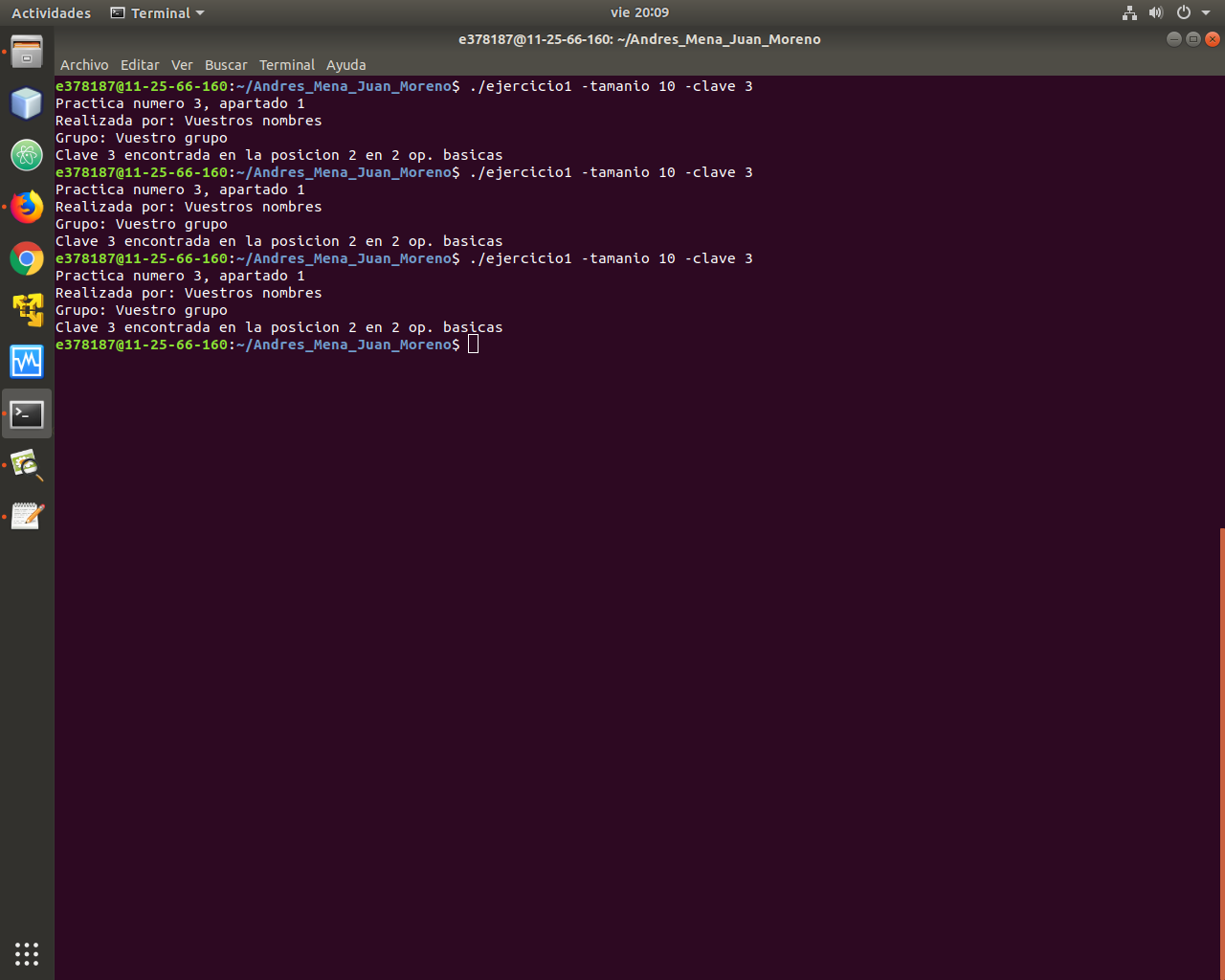
**5. Resultados, Gráficas**

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

5.1 Apartado 1

Primero, ejecutamos usando búsqueda lineal sobre tablas no ordenadas:

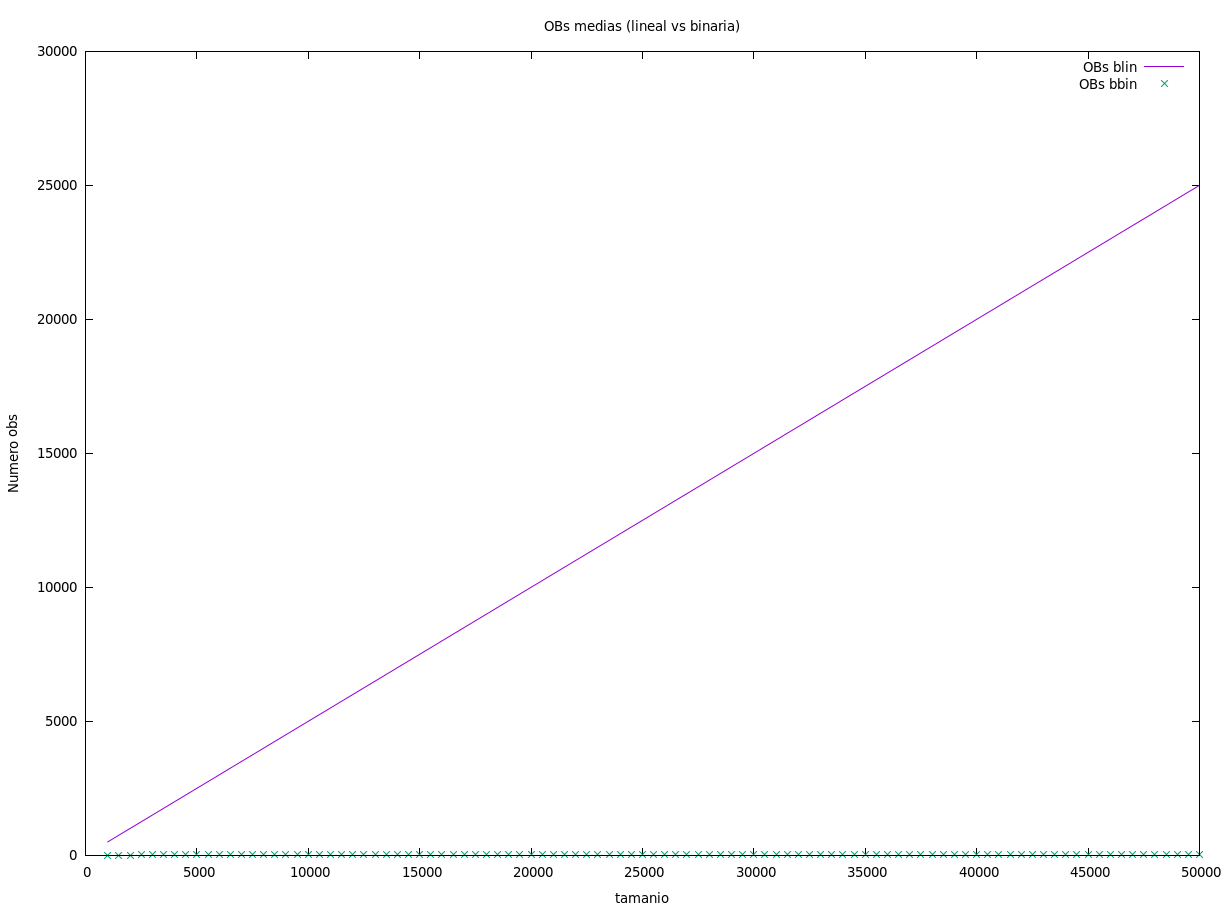
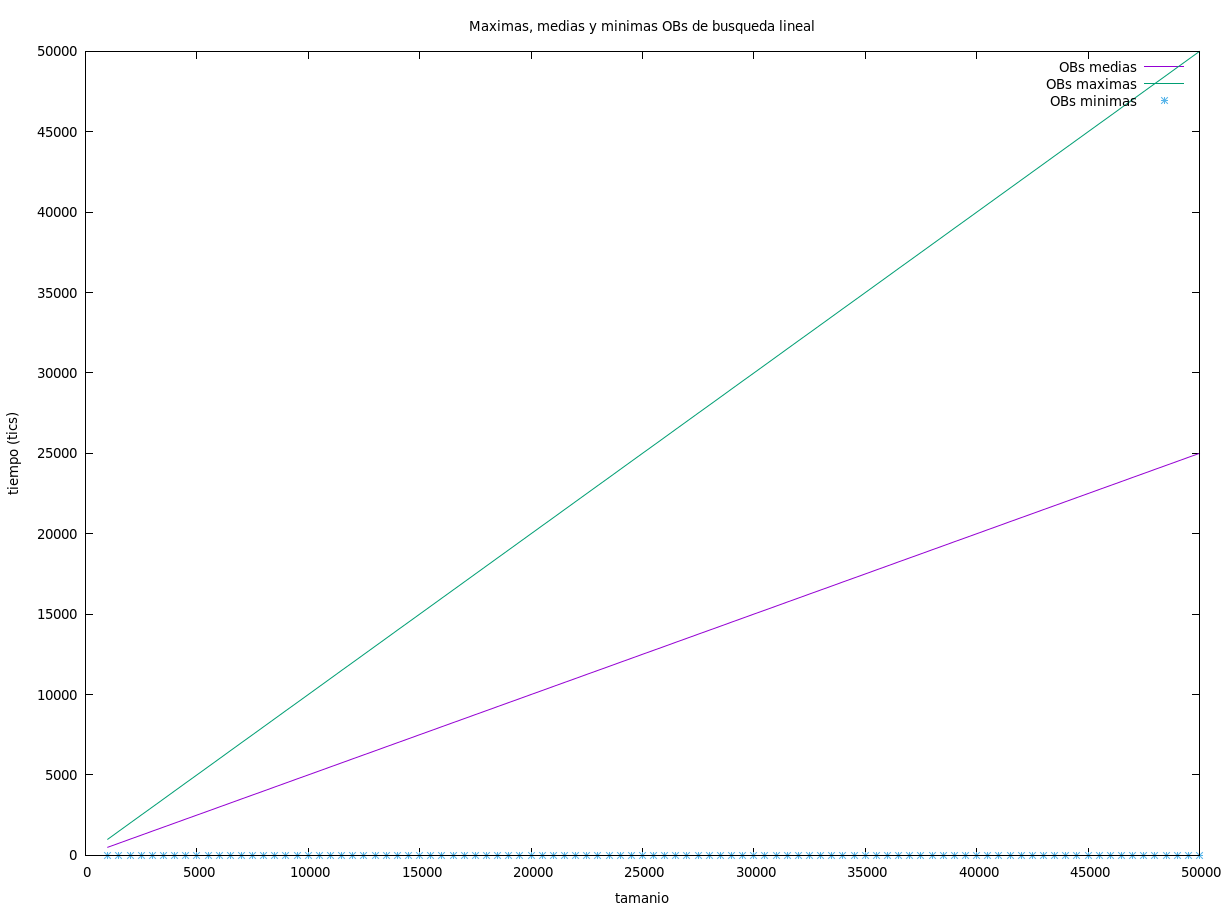
Como podemos observar, el numero de operaciones básicas va cambiando, y esto es porque cada vez que ejecutamos el programa, se crea una permutación diferente aleatoria, por lo que el elemento buscado no se encuentra siempre en la misma posición.

A continuación, ejecutamos usando búsqueda binaria sobre tablas ordenadas:

Como podemos observar, siempre realiza el mismo número de operaciones básicas (comparaciones de clave), ya que las tablas que estamos creando siempre son iguales, y por lo tanto el algoritmo siempre realiza el mismo procedimiento.

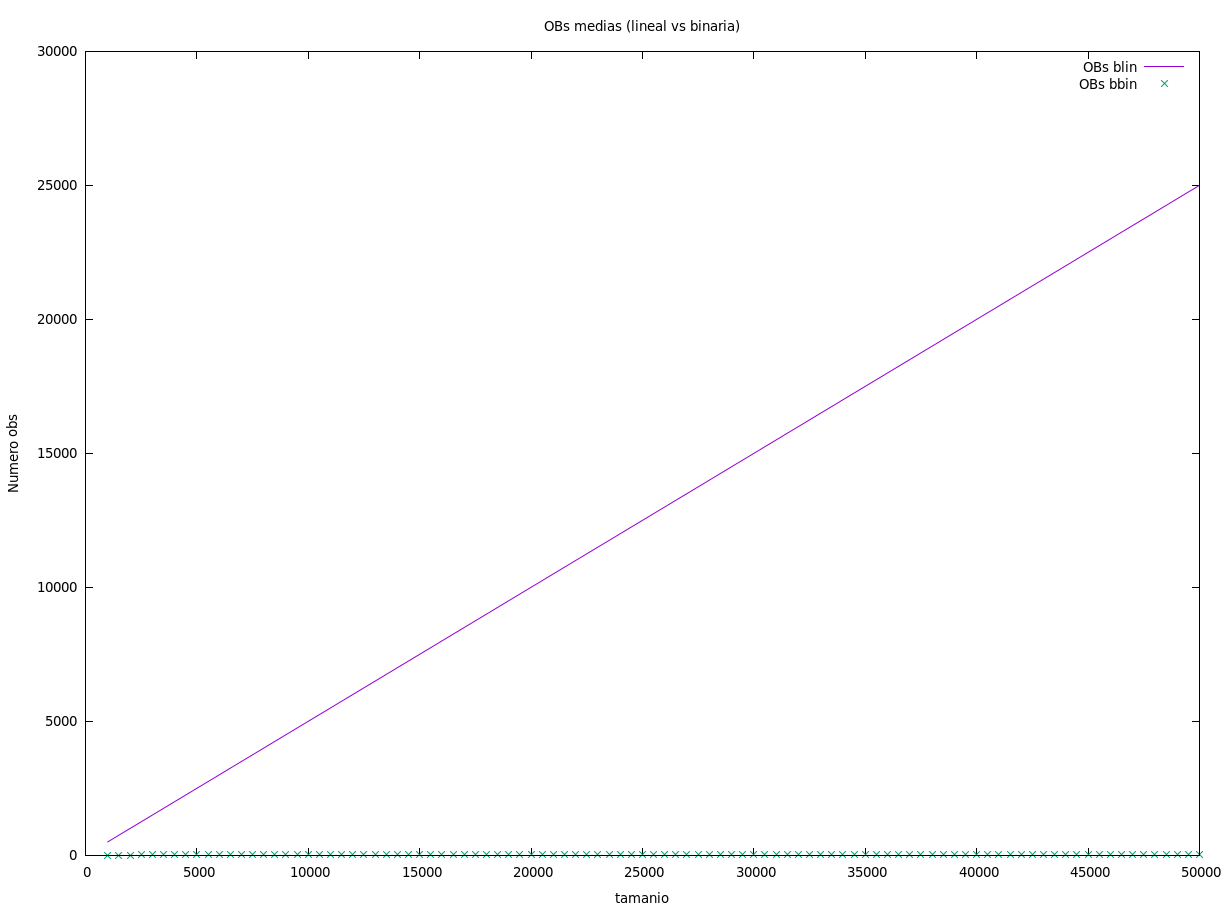
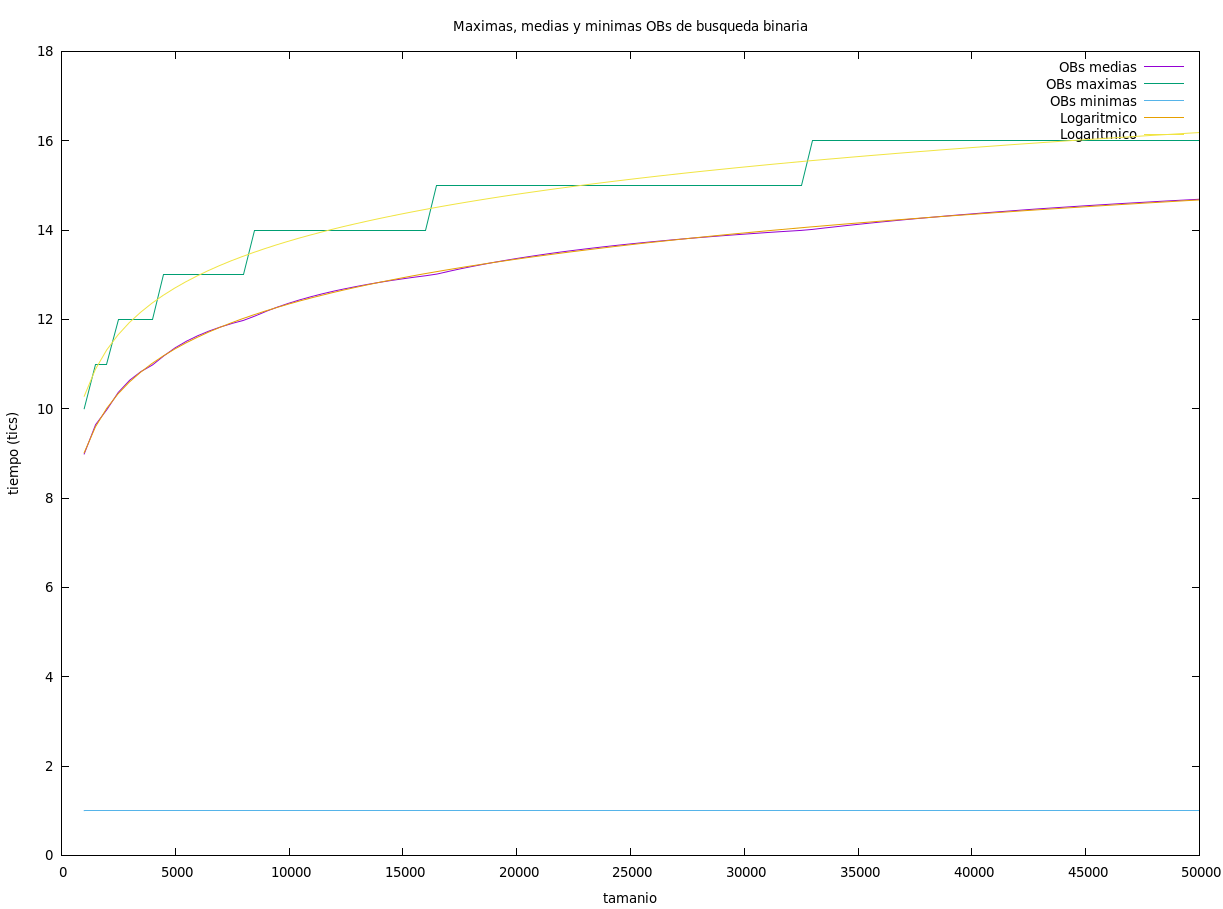
5.2 Apartado 2

Gráfica con el número máximo, mínimo y medio de OBs de la búsqueda lineal.

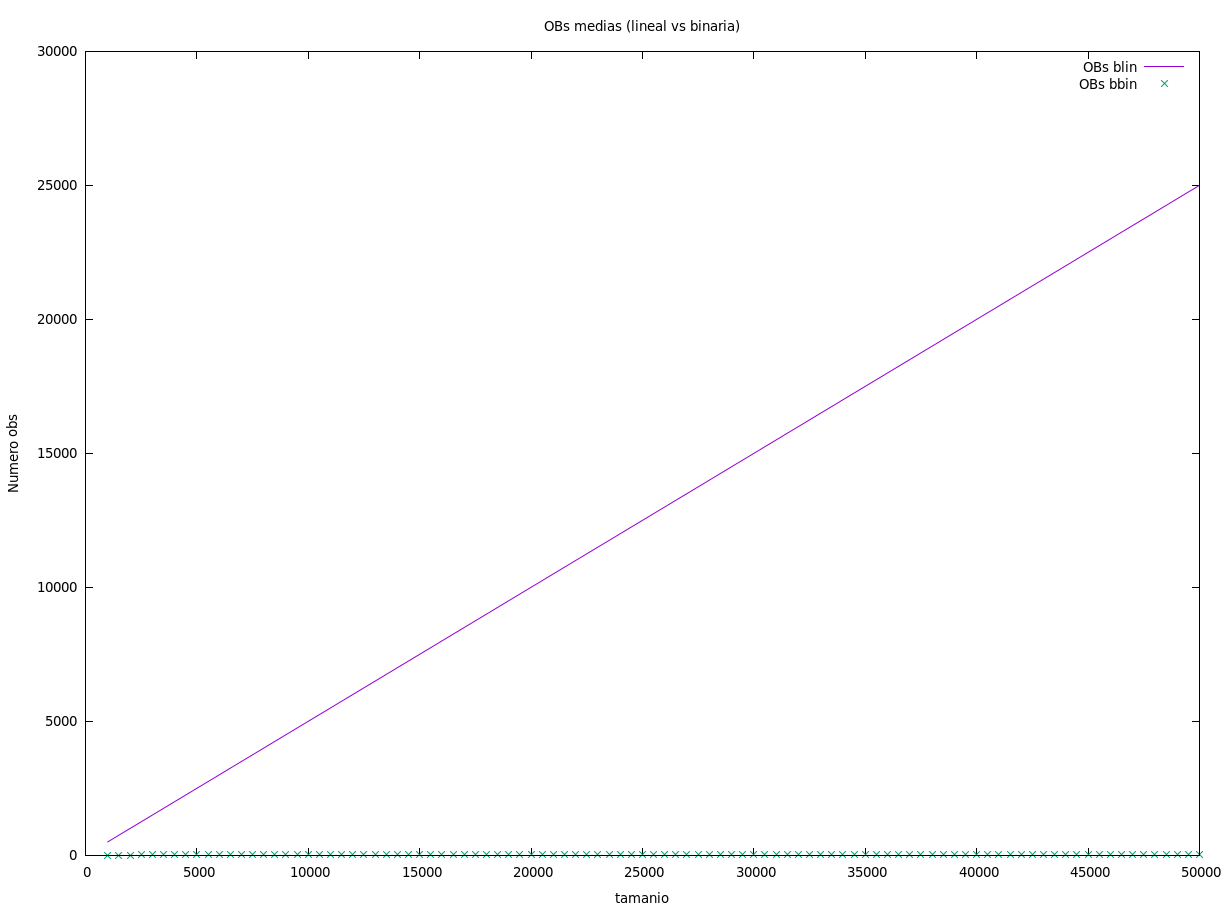


Podemos observar que las trayectorias de esta gráfica se tratan de líneas rectas, y esto es debido a que estamos usando el algoritmo de búsqueda lineal. El caso mejor de este algoritmo es que el elemento buscado se encuentre en la primera posición de nuestra tabla, sin embargo, esto ocurre con muy poca frecuencia, y si observamos el resto de los casos (medio y peor), vemos que crece de forma lineal en función del tamaño de la tabla.

Gráfica con el número máximo, mínimo y medio de OBs de la búsqueda binaria.

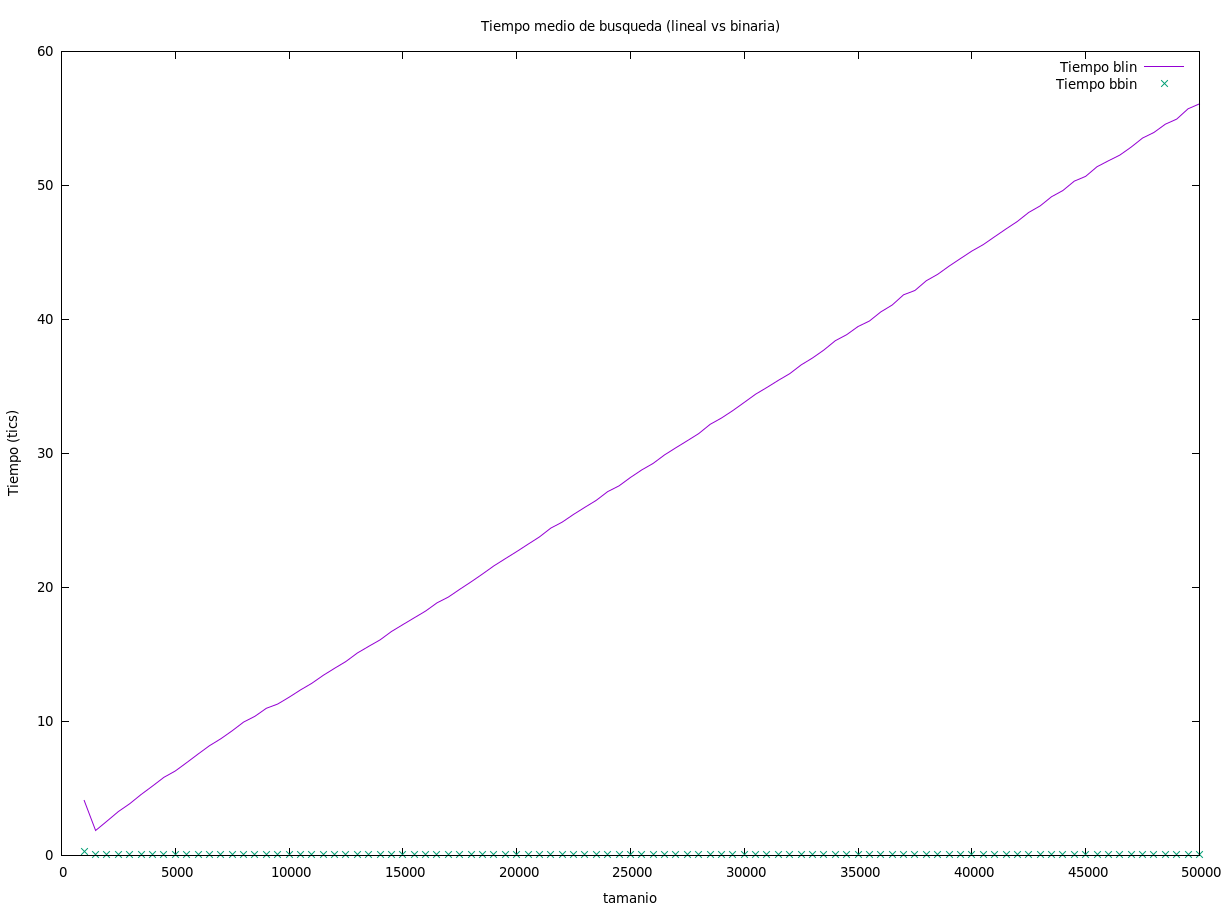


Podemos observar que al igual que en la búsqueda lineal, el caso mejor se trata de una línea recta en el 1, que sería encontrar el elemento justo en el medio de nuestra tabla. Por otra parte, los rendimientos medios y peor siguen trayectorias logarítmicas, que como podemos observar en la leyenda de la gráfica, hemos podido ajustar una recta a ellas. La forma de la curva del caso peor es peculiar, y para explicar su forma podemos usar la idea de un árbol: El caso peor tiene lugar cuando el elemento se encuentra en el ultimo nivel, pero el último nivel puede tener varios nodos, por lo que todos ellos nos darían el rendimiento peor. Como podemos observar, cada vez la zona horizontal de esa curva es el doble de larga, y esto ocurre cada vez que nuestro árbol tiene un nivel más de profundidad y es capaz de albergar el doble de nodos en su último nivel.

Gráfica comparando el número promedio de OBs entre la búsqueda lineal y la búsqueda binaria, comentarios a la gráfica.

En esta gráfica podemos ver claramente la gran diferencia entre los dos algoritmos, ya que al tener en un mismo gráfico el número de operaciones básicas que realiza la búsqueda binaria y las que la realiza la búsqueda lineal, vemos claramente como el numero de comparaciones que se ejecutan en bbin es insignificante al compararlo con las de blin.

En esta gráfica, podríamos asemejar una línea recta al número de obs de la búsqueda lineal, que quedaría justo detrás de la recta que hay en la gráfica, y una función logarítmica a la búsqueda binaria, que iría por la parte inferior de la gráfica, que a simple vista parecería una recta debido a la escala utilizada en la representación.

Gráfica comparando el tiempo promedio de reloj entre la búsqueda lineal y la búsqueda binaria, comentarios a la gráfica.

Al igual que en la gráfica anterior, este es otro ejemplo de la gran diferencia que hay entre el rendimiento de la búsqueda binaria y la búsqueda lineal. Para la realización de esta gráfica, la unidad de tiempo utilizada han sido los tics, ya que, si intentamos convertir esta medida a segundos, la búsqueda binaria nos da como resultado 0.000000, debido a ser un algoritmo que encuentra lo que busca muy rápido.

En esta gráfica, podríamos asemejar una línea recta al tiempo medio de la búsqueda lineal, que quedaría justo detrás de la recta que hay en la gráfica, y una función logarítmica a el tiempo de la búsqueda binaria, que iría por la parte inferior de la gráfica, que a simple vista parecería una recta debido a la escala utilizada en la representación.

**6. Respuesta a las preguntas teóricas.**

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

6.1 Pregunta 1

La operación básica tanto de la búsqueda lineal como la búsqueda binaria es la comparación de clave que se realiza.

En la búsqueda binaria, se realiza una comparación del elemento buscado con el elemento que se encuentra en la mitad de la tabla que buscamos. En caso de que ese elemento sea el correcto, ya hemos encontrado nuestra clave. Si no es este, comprobamos si es mayor o menor, y cogemos la parte a la derecha o izquierda del array respectivamente, repitiendo el proceso de partir y comprobar si encontramos el elemento de forma iterativa.

En la búsqueda lineal, vamos recorriendo el array elemento a elemento desde el comienzo de este, y comprobando si la posición en la que nos encontramos es el elemento que estamos buscando. Finaliza cuando llegamos al final de nuestro array.

6.2 Pregunta 2

Para la búsqueda lineal, el caso peor es **WBlin(N)=N**. Esto ocurre cuando la clave que buscamos es la última del diccionario. El caso peor de **BBlin(N)=1** y ocurre cuando la clave que buscamos es la primera.

El caso peor de la búsqueda binaria es **WBbin(N)=log(N)+O(1).** Ocurre para las claves que ocupan determinadas posiciones en el diccionario. Las posiciones y la cantidad de ellas dependen del tamaño del diccionario. El caso mejor de **BBbin(N)=1**, ya que ocurre cuando se busca la clave que está en la mitad de la tabla.

6.3 Pregunta 3

Si la manera con la que generamos claves es una distribución no uniforme, podemos confirmar que el orden de ejecución medio es O(1). Esto es porque las claves que se busquen serán las que estén en las primeras posiciones del diccionario y por lo tanto se encontraran con costes O(1). Alguna clave puede que tenga un coste mayor, pero la media se mantendrá en O(1).

6.4 Pregunta 4

Búsqueda lineal: Es un método que busca bien, ya que lo que hace es ir recorriendo toda la tabla que le introducimos. La tabla introducida no tiene que estar ordenada necesariamente. Comenzamos en el primer elemento de la tabla, lo comparamos con el parámetro recibido y en caso de coincidencia devolvemos la posición en la que se encuentra el elemento, y cuantas comparaciones se han realizado hasta llegar a esa comparación exitosa. En caso de no coincidencia, examinaremos el siguiente elemento, realizando el mismo procedimiento, y así sucesivamente hasta recorrer toda la tabla. En caso de terminar la tabla y que no hayamos tenido ninguna coincidencia, tendremos que devolver cuantas comparaciones de clave se han llevado a cabo y devolver que el elemento buscado no se ha encontrado (ERR).

Búsqueda binaria: Es un método que funciona cuando actúa sobre tablas ordenadas. Lo que hace en un primer momento, es “partir” la tabla por la mitad, y comprueba si el elemento buscado es el que se encuentra justo en medio. En caso de ser así, devuelve la posición del elemento y el número de comparaciones básicas que se han realizado. Si no es así, lo que se hace es comparar si el elemento que queremos buscar es mayor o menor que el elemento que se encontraba en el medio. En caso de ser mayor, nos quedamos con la mitad superior de la tabla, y realizamos el mismo proceso de partir esta nueva tabla por la mitad y comparar si el elemento buscado es el que se encuentra justo en el centro. Análogamente, haríamos este proceso con la mitad inferior en caso de que el elemento buscado fuese menor. El proceso finaliza cuando se encuentra el elemento buscado (devolviendo su posición y cuantas comparaciones de claves se han realizado), o cuando ya no podemos partir más la tabla porque ya solo nos queda un elemento, lo que significa que nuestro elemento buscado no está en la tabla.

Este método no funcionaría con tablas no ordenadas, y sí que funciona correctamente sobre tabla ordenadas ya que sabemos con certeza en que mitad de la tabla se debería encontrar nuestro elemento con una comparación de clave.

**7. Conclusiones finales.**

En esta práctica hemos implementado un diccionario de claves, con el objetivo de probar los algoritmos de búsqueda lineal y búsqueda binaria. Para ello, crearemos diccionarios que pueden estar tanto ordenados (para probar la búsqueda binaria) como desordenados (para probar la búsqueda lineal). En estos diccionarios podemos insertar elementos de forma masiva (usando arrays) o de uno en uno. Dependiendo de si es un diccionario ordenado o desordenado, la inserción seguirá un código u otro, ya que, si el diccionario está ordenando, en cada inserción tendremos que ir recorriendo el diccionario para colocar el elemento en la situación correcta.

Como hemos podido observar a lo largo de la práctica, el método de búsqueda binaria es mejor que el método de búsqueda lineal para todos los casos. Esto es porque en la búsqueda lineal no estamos aplicando ningún algoritmo, por lo que tenemos que ir recorriendo todas las posiciones del array, y la velocidad con la que encontremos una clave dependerá de la aleatoriedad de la tabla introducida. Por otra parte, el método de búsqueda binaria se basa en la idea de que el array introducido esta ordenado, y por cada comparación de clave es capaz de decir en qué mitad de la tabla se encontraría el elemento que buscamos (en caso de estar). De esta forma, estamos “deshaciéndonos” de la mitad de los elementos de la tabla con cada comparación de clave, por lo que es un algoritmo de búsqueda que busca mucho más rápido y que encuentra de forma más eficaz tanto para caso medio como para caso peor.

Cabe destacar de los dos algoritmos tienen el mismo caso mejor, que es 1 sola operación básica. Esto tiene lugar en búsqueda lineal cuando el elemento se encuentra en la primera posición de nuestro array, mientras que en búsqueda binaria tiene lugar cuando se encuentra en el punto medio de nuestro array. Este caso es poco común, ya que depende del elemento que se está buscando, y de la tabla que tenemos, por lo que no es un caso que sirva para determinar qué forma de búsqueda es mejor.

También hay que tener en cuenta que, a la hora de introducir una tabla para realizar una búsqueda binaria sobre ella, hemos tenido que invertir un coste en la ordenación de la tabla, lo que supone un inconveniente de búsqueda binaria frente a búsqueda lineal.