Análisis de Algoritmos 2016/2017

Práctica 3

Alfonso Villar y Víctor García, Grupo 12

Código	Gráficas	Memoria	Total

1. Introducción.

En esta última práctica de la asignatura de Análisis de Algoritmos estudiamos los algoritmos de búsqueda BBIN (búsqueda binaria), BLIN (búsqueda lineal) y BLIN_AUTO (búsqueda lineal autorganizada) sobre diccionarios. Para ello, determinamos experimentalmente tanto los tiempos medios como las operaciones básicas de sendas búsquedas sobre diccionarios que emplean como tipo de datos una tabla para el posterior estudio de los resultados obtenidos, realizando diversas gráficas y extrayendo conclusiones de las mismas.

2. Objetivos

2.1 Apartado 1

En este primer ejercicio creamos el TAD diccionario, previamente definido en en fichero busqueda.h, utilizando una tabla de enteros (ordenada o desordenada) como tipo abstracto de datos. Posteriormente, implementamos en busqueda.c todas las primitivas del TAD diccionario, además de las funciones de búsqueda bbin, blin y blin_auto.

2.2 Apartado 2

En este apartado implementamos en tiempos.c una serie de funciones para realizar medidas del rendimiento de las funciones de búsqueda implementadas anteriormente, tanto en lo que respecta al tiempo promedio de ejecución como a las operaciones básicas (promedio, máximas y mínimas) de las búsquedas.

3. Herramientas y metodología

3.1 Apartado 1

Entorno de desarrollo: Sublime Text

Entorno de ejecución: Linux

Comprobación de pérdidas de memoria y violaciones de segmento: Valgrind

Compilación del código: gcc -ansi -pedantic Flags: -Wall -g

3.2 Apartado 2

Entorno de desarrollo: Sublime Text

Entorno de ejecución: Linux

Elaboración de gráficas: Gnuplot

Comprobación de pérdidas de memoria y violaciones de segmento: Valgrind

Compilación del código: gcc -ansi -pedantic Flags: -Wall -g

4. Código fuente

4.1 Apartado 1

```
/*
*  Funcion: ini_diccionario
*                                   Esta funcion inicia el diccionario
*              con un tamaño fijado y en un estado ordenado o desordenado
*/
PDICC ini_diccionario (int tamanio, char orden)
{
        if (tamanio < 1 || (orden != ORDENADO && orden != NO_ORDENADO))
                return NULL;

        PDICC dic = (PDICC) malloc (sizeof (DICC));
        if (dic == NULL)
                return NULL;

        dic->tamanio = tamanio;
        dic->n_datos = 0;
        dic->orden = orden;
        dic->tabla = (int *) malloc (sizeof (int)*tamanio);
        if (dic->tabla == NULL){
                free (dic);
                return NULL;
        }
        return dic;
}


/*
*  Funcion: libera_diccionario
*              Esta funcion libera el diccionario
*/
void libera_diccionario(PDICC pdicc)
{
        assert (pdicc != NULL);
```

```
	free (pdicc->tabla);
	free (pdicc);
	return;
}


/*
*  Funcion: inserta_diccionario
*		Esta funcion inserta una clave en el diccionario
*		si esta ordenado lo coloca ordenado y si no al final de todo
*/
int inserta_diccionario(PDICC pdicc, int clave)
{
	int cont = 0;
	assert (pdicc != NULL && clave >= 0);

	if (pdicc->tamanio == pdicc->n_datos)
		return ERR;

	if (pdicc->orden == NO_ORDENADO){ /*Si el diccionario no es ordenado*/
		pdicc->tabla[pdicc->n_datos] = clave;
		pdicc->n_datos++;
	}

	else{ /*Si el diccionario es ordenado*/
		pdicc->tabla[pdicc->n_datos] = clave;
		pdicc->n_datos++;
		int last = pdicc->tabla[pdicc->n_datos-1];
		int j = pdicc->n_datos-2;

		while (j >= 0 && pdicc->tabla[j] > last){
			pdicc->tabla[j+1] = pdicc->tabla[j];
			j--;
			cont++;
		}
		pdicc->tabla[j+1] = clave;
```

```
        }
        return cont;
}


/*
*  Funcion: isercion_masiva_diccionario
*           Esta funcion inserta masivamente claves en el diccionario
*           utilizando en bucle inserta_diccionario
*/
int insercion_masiva_diccionario (PDICC pdicc,int *claves, int n_claves)
{
        assert (pdicc != NULL && claves != NULL && n_claves >= (pdicc->tamanio - pdicc->n_datos) && n_claves >= 1);
        int i;
        for (i = 0; i < n_claves; i++){
                if (inserta_diccionario(pdicc, claves[i]) == ERR)
                        return ERR;
        }
        return OK;
}


/*
*  Funcion: busca_diccionario
*           Esta funcion busca una clave en el diccionario y
*           hará que ppos apunte al numero de la posicion de la clave
*           utilizando un metodo de búsqueda definido por método
*/
int busca_diccionario(PDICC pdicc, int clave, int *ppos, pfunc_busqueda metodo)
{
        int cont;
        assert(pdicc != NULL && clave >= 0);

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

```
/*
*  Funcion: imprime_diccionario
*           Esta funcion imprime el diccionario
*/
void imprime_diccionario(PDICC pdicc)
{
        assert(pdicc != NULL);
        int i;
        for (i = 0; i < pdicc->n_datos; i++){
                printf ("%d\t", pdicc->tabla[i]);
        }
        return;
}


/* Funciones de busqueda del TAD Diccionario */

/*
*  Funcion: bbin
*           Esta es una funcion de busqueda en la cual
*           se reduce el tamaño de la tabla a la mitad en cada cdc
*/
int bbin(int *tabla, int P, int U, int clave, int *ppos)
{
        assert (tabla != NULL && clave >= 0);
        int m = (P + U) / 2;

        if (P > U) return NO_ENCONTRADO;

        if (tabla[m] == clave){
                *ppos = m;
                return 1;
        }

        if (clave < tabla[m]){
                return 1 + bbin(tabla, P, m-1, clave, ppos);
```

```
        }

        if (clave > tabla[m]){
                return 2 + bbin(tabla, m+1, U, clave, ppos);
        }

        return NO_ENCONTRADO;
}


/*
*  Funcion: blin
*           Esta es una funcion de busqueda en la cual
*           se busca de 1 en 1 por la tabla la clave linealmente
*/
int blin(int *tabla,int P,int U,int clave,int *ppos){
        assert (tabla != NULL && P <= U && clave >= 0);

        int i;
        int cont = 0;

        for (i = P; i <= U; i++, cont++){
                if (tabla[i] == clave){
                        *ppos = i;
                        cont++;
                        return cont;
                }
        }
        return NO_ENCONTRADO;
}

/*
*  Funcion: blin_auto
*           Esta es una funcion de busqueda en la cual
*           se busca de 1 en 1 por la tabla la clave linealmente
*           y que cuando encuentra la clave hace un swap hacia delante
*/
```

```
int blin_auto(int *tabla,int P,int U,int clave,int *ppos){
	assert (tabla != NULL && P <= U && clave >= 0);
	int i;
	int cont = 0;

	for (i = P; i <= U; i++, cont++){
		if (tabla[i] == clave){
			if (i == P){
				*ppos = i;
				cont++;
				return cont;
			}
			tabla[i] = tabla [i-1];
			tabla[i-1] = clave;
			*ppos = i-1	;
			return cont;
		}
	}
	return NO_ENCONTRADO;
}
```

4.2 Apartado 2

```
/*****************************************************/
/* Funcion: genera_tiempos_busqueda Fecha:09/12/16 */
/* Autores: Alfonso Villar y Víctor García	*/
/*                                      */
/* Entradas: fichero!=NULL & 0<num_min<=num_max &  */
/* incr>0 & n_veces>0                          */
/*
*/
/* pfunc_busqueda metodo: introduce el metodo de   */
/* busqueda a usar                             */
/* int n_veces: veces a ejecutar cada busqueda     */
/* int incr: incremento del tamaño de la tabla     */
/* int num_max: tamaño máximo de la tabla          */
/* int num_min:tamaño mínimo de la tabla           */
/* int fichero:fichero donde guardar los resultados*/
```

```
/* int orden: indica si la tabla esta o no ordenada*/
/* pfunc_generador_claves:introduce el metodo de   */
/* generacion de claves                            */
/*
*/
/* Salida:                                         */
/* Devuelve ERR si hay error u OK si no            */
/*                                                 */
/***************************************************/

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 i;
        int iteraciones;
        iteraciones = (int)((num_max-num_min)/(incr))+1;
        PTIEMPO ptiempo;
        ptiempo = (PTIEMPO) malloc (iteraciones * (sizeof (TIEMPO)));
        if (ptiempo == NULL)
                return ERR;

        for (i = 0; i < iteraciones; i++){
                if (tiempo_medio_busqueda(metodo, generador, orden, num_min+i*incr,
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;
}
```

```
/***************************************************/
/* Funcion: tiempo_medio_busqueda Fecha:09/12/16   */
/* Autores: Alfonso Villar y Víctor García         */
/*                                                 */
/* Entradas: 0<num_min<=num_max & n_veces>0        */
/*
*/
/* pfunc_busqueda metodo: introduce el metodo de   */
/* busqueda a usar                                 */
/* int n_veces: veces a ejecutar cada busqueda     */
/* int num_max: tamaño máximo de la tabla          */
/* int num_min:tamaño mínimo de la tabla           */
/* int orden: indica si la tabla esta o no ordenada*/
/* pfunc_generador_claves:introduce el metodo de   */
/* generacion de claves                            */
/*                                                 */
/* Salida:                                         */
/* Devuelve ERR si hay error u OK si no            */
/*                                                 */
/***************************************************/

short tiempo_medio_busqueda(pfunc_busqueda metodo, pfunc_generador_claves
generador, int orden, int n_claves, int n_veces, PTIEMPO ptiempo){
        int ob = 0;
        int i, pos;
        int medio_ob = 0;
        int max_ob = 0;
        int min_ob = INT_MAX;
        clock_t ini, fin;

        PDICC pdicc = ini_diccionario (n_claves, orden);
        if (pdicc == NULL) return ERR;

        int* perm = genera_perm(n_claves);
        if (perm == NULL){
                libera_diccionario (pdicc);
                return ERR;
```

```
}

if (insercion_masiva_diccionario(pdicc, perm, n_claves) == ERR){
        libera_diccionario (pdicc);
        free (perm);
        return ERR;
}

int* tabla = (int*) malloc (sizeof (int) * n_claves * n_veces);
if (tabla == NULL){
        libera_diccionario (pdicc);
        free (perm);
        return ERR;
}

generador (tabla, n_claves*n_veces, n_claves);

ini = clock();
for (i = 0; i < n_claves*n_veces; i++){
        ob = busca_diccionario(pdicc, tabla[i], &pos, metodo);
        if (ob == ERR)
                return ERR;
        if (max_ob < ob) /*Comparamos OB iterativamente con max_ob para
                                        actualizar el numero maximo de OB*/
                max_ob = ob;
        if (min_ob > ob) /*Comparamos OB iterativamente con min_ob para
                                        actualizar el numero minimo de OB*/
                min_ob = ob;
        medio_ob += ob;

}
fin = clock();
ptiempo->n_perms = n_claves;
ptiempo->tamanio = n_claves;
ptiempo->medio_ob = (double) medio_ob / n_claves / n_veces;
ptiempo->max_ob = max_ob;
ptiempo->min_ob = min_ob;
```

```
        ptiempo->tiempo = (double) (fin - ini) / n_claves / n_veces /
CLOCKS_PER_SEC;
        ptiempo->n_veces = ob;


        libera_diccionario (pdicc);
        free (tabla);
        free (perm);
        return OK;
}
```

5. Resultados, Gráficas

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

5.1 Apartado 1

En este apartado comprobamos, con el ejercicio1, las funciones de búsqueda bbin, blin y blin_auto, respectivamente. Las salidas de la terminal al ejecutar sendas búsquedas son las siguientes (en el orden expuesto previamente).

```
usuario@usuario-Satellite-Pro-A50-C:~/segundo/ANAL/practica3$ ./ejercicio1 -tamanio 100 -clave 25
Practica numero 3, apartado 1
Realizada por: Alfonso Villar y Víctor García
Grupo: 12
Clave 25 encontrada en la posicion 24 en 2 op. basicas
usuario@usuario-Satellite-Pro-A50-C:~/segundo/ANAL/practica3$
```

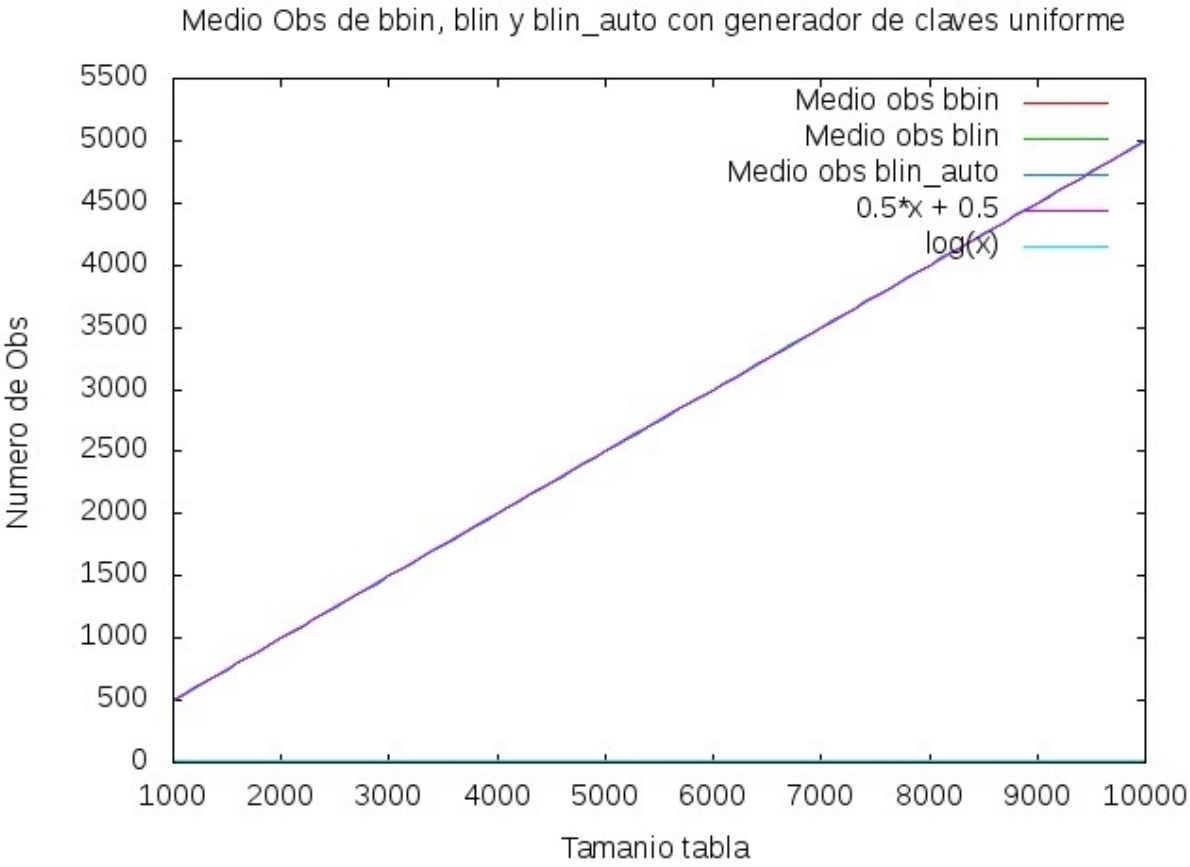
```
usuario@usuario-Satellite-Pro-A50-C:~/segundo/ANAL/practica3$ ./ejercicio1 -tamanio 100 -clave 25
Practica numero 3, apartado 1
Realizada por: Alfonso Villar y Víctor García
Grupo: 12
Clave 25 encontrada en la posicion 58 en 59 op. basicas
usuario@usuario-Satellite-Pro-A50-C:~/segundo/ANAL/practica3$
```

```
usuario@usuario-Satellite-Pro-A50-C:~/segundo/ANAL/practica3$ ./ejercicio1 -tamanio 100 -clave 25
Practica numero 3, apartado 1
Realizada por: Alfonso Villar y Víctor García
Grupo: 12
Clave 25 encontrada en la posicion 20 en 21 op. basicas
usuario@usuario-Satellite-Pro-A50-C:~/segundo/ANAL/practica3$
```

Notas: La posición indicada está dada “en el array”, es decir, la primera posición es la 0 y la última posición es N-1 (siendo N el tamaño de la tabla). Además cabe mencionar que en bbin la búsqueda se hace (obviamente) sobre un diccionario ordenado, mientras que blin y blin_auto se realizan sobre diccionarios desordenados. La comprobación del correcto funcionamiento de estas funciones se ha comprobado, además, imprimiendo la tabla y viendo los resultados.

5.2 Apartado 2

En este apartado llevamos a cabo diversas gráficas. La primera de ellas representa el número promedio de Obs de bbin, blin y blin_auto con un generador de claves uniforme:

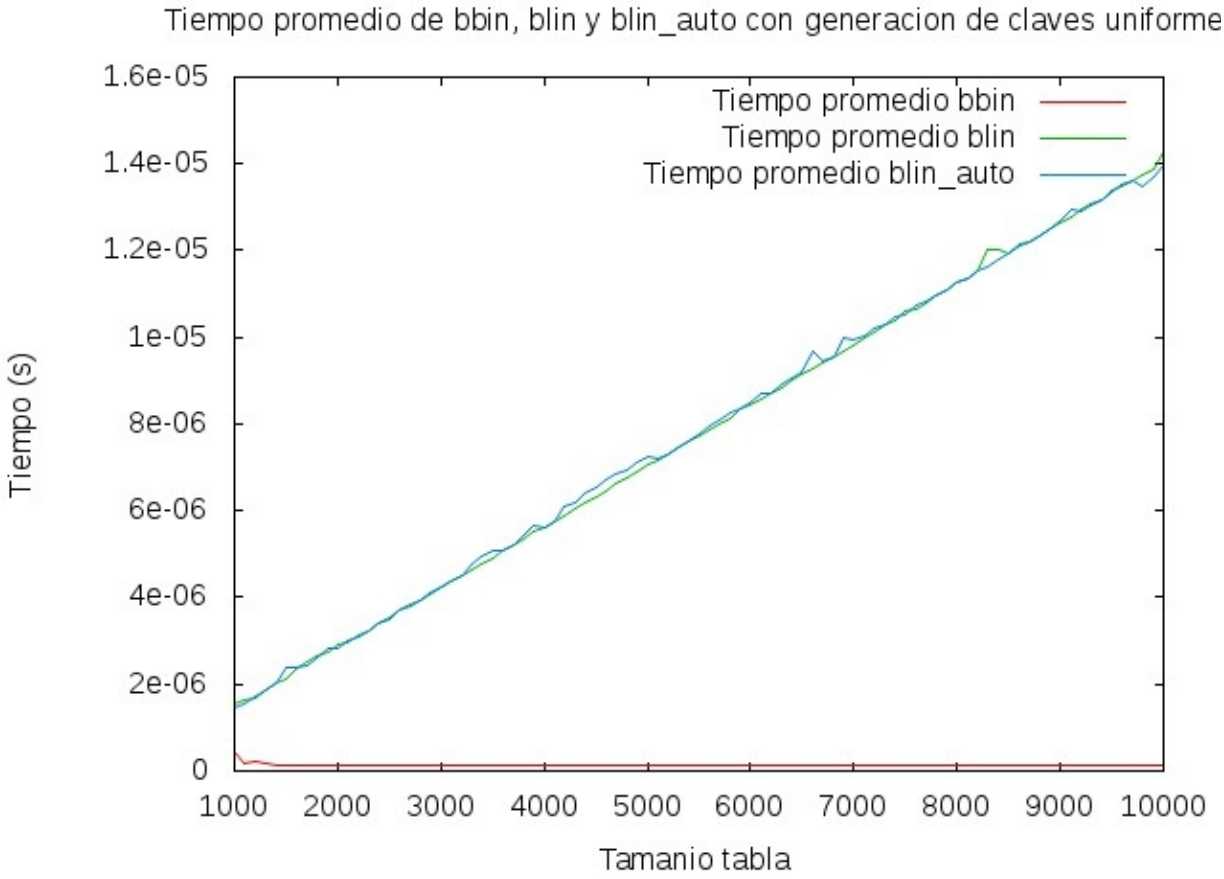


En el caso de que blin_auto se realice a partir de un generador de claves uniforme, podemos ver que su número medio de Obs es igual al de blin (n_veces=1). En la gráfica se ha aproximado blin y bbin. Podemos observar que en blin, su gráfica de medio Obs es lineal, siendo aproximada por la recta de color morado (que como muestran sus valores, esta recta de aproximación es lineal) que coincide exactamente con la misma, así como con blin_auto. En el caso de bbin, podemos observar que es la que menos Obs realiza, lo cual coincide con lo estudiado teóricamente de que realiza de media log(N) comparaciones de claves. Además, esto se ve reforzado al ajustar la gráfica de bbin con una funcion logáritmica (representada en la gráfica con la recta de color azul claro) y ver que coincide exactamente con la misma. Las aproximaciones realizadas con blin y bbin son tan exactas a las mismas que apenas se aprecian las gráficas de las mismas.

En el caso de representar las Obs promedio de bbin, blin y blin_auto con un generador de claves potencial, las Obs promedio de bbin apenas cambian respecto a las obtenidas con el generador de claves uniforme, y seguirían estando los valores promedio de Obs muy por debajo de blin y blin_auto (leer siguiente linea). Para que las Obs promedio de blin_auto cambiaran y dejaran de ser prácticamente las mismas que

blin, deberíamos aumentar el número de veces que se busca cada clave (por ejemplo a 1000), para que la autorganización establecida en este algoritmo de búsqueda permitiera que el número promedio de Obs fuera mucho menor (así como el tiempo de ejecución). Para ver si, en este caso, el número de obs promedio de blin_auto es mayor o menor que el de bbin, mirar las gráficas 3 y 4.

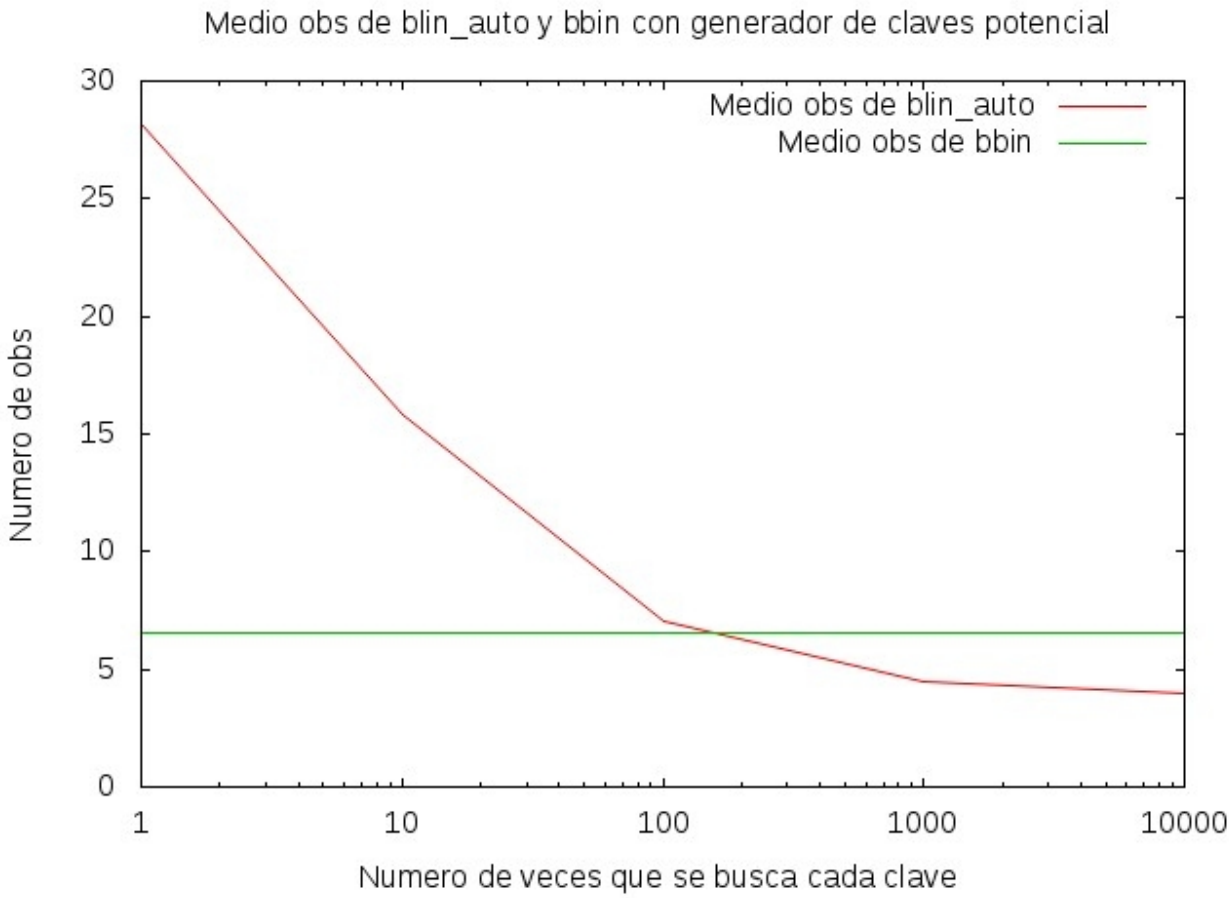
La siguiente gráfica(2) corresponde al tiempo medio de reloj de bbin, blin y blin_auto en el caso de utilizar un generador de claves uniforme:



Como podemos observar en la gráfica, al tratarse de un generador de claves uniforme, el tiempo promedio de blin_auto (n_veces=1) coincide con el de blin, siendo ambos tiempos bastante superiores al tiempo promedio de bbin. Esto coincide con lo esperado, pues bbin es mucho más rápido que las búsquedas lineales.

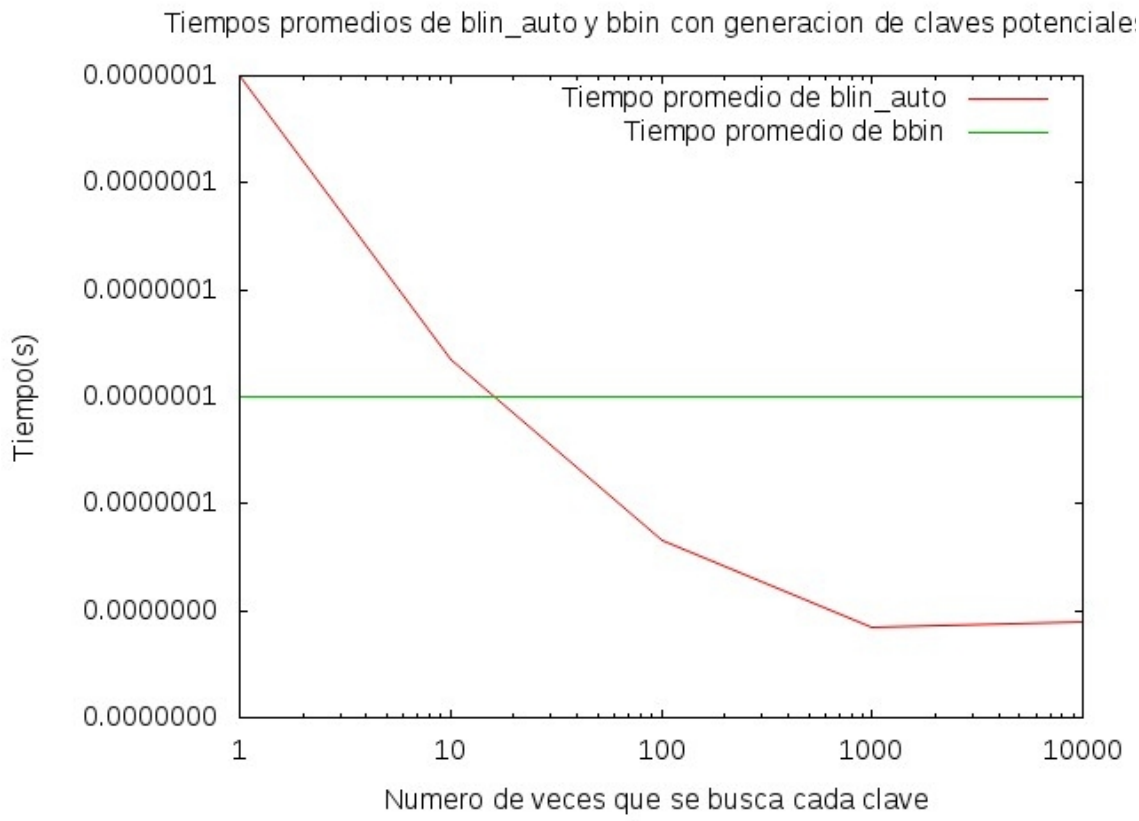
En el caso de que utilizáramos un generador de claves potencial, el tiempo promedio de bbin y blin no sufriría grandes cambios. Sin embargo, en lo que respecta a blin_auto, si aumentamos n_veces a valores como 1000, al autorganizarse el diccionario, el tiempo promedio que tarda en encontrar una clave desciende notablemente. Para ver si, en este caso, el tiempo promedio de blin_auto es mayor o menor que el de bbin, mirar las gráficas 3 y 4.

La siguiente gráfica(3) representa el número promedio de Obs de blin_auto en diccionarios de tamaño fijo (de 100 elementos) en función del valor de n_veces, es decir, en función de cuántas veces se busque una clave. Además, se representa el número promedio de Obs de bbin para el caso de un diccionario de 100 elementos:



Como se puede observar en la gráfica, a medida que aumenta el número de veces que se busca cada clave en el diccionario, al autorganizarse el diccionario, el número medio de Obs desciende, hasta el punto de que llega a ser menor que el número medio de Obs de bbin a partir de las 100 comparaciones (en este caso). La gráfica de medio obs de blin_auto se encuentra entre una gráfica logarítmica y una gráfica lineal. Esto demuestra que, en ciertos casos, la blin_auto es más eficiente que bbin, la cual pensabamos en primera instancia que no se podía mejorar.

Finalmente, en la última gráfica(4) se representa el tiempo promedio de reloj de blin_auto en diccionarios de tamaño fijo (de 100 elementos) en función del valor de n_veces, es decir, en función de cuántas veces se busque una clave. Además, se representa el tiempo promedio de bbin para el caso de un diccionario de 100 elementos:



Como podemos observar, a medida que aumenta el número de veces que se busca cada clave en el diccionario, al autorganizarse el diccionario, el tiempo promedio de la búsqueda lineal autorganizada desciende, hasta el punto de que llega a ser menor que el tiempo promedio de bbin alrededor de las 20 comparaciones (en este caso). La gráfica del tiempo promedio de blin_auto se encuentra entre una gráfica logarítmica y una gráfica lineal. Esto demuestra que, en ciertos casos, la blin_auto es más eficiente que bbin, la cual pensabamos en primera instancia que no se podía mejorar.

5. Respuesta a las preguntas teóricas.

5.1 ¿Cúal es la operación básica de bbin, blin y blin auto?

La operación básica de bbin, blin y blin_auto es la comparación de clave.

5.2 Dar tiempos de ejecución en función del tamaño de entrada n para el caso peor WSS(n) y el caso mejor BSS(n) de BBin y BLin. Utilizar la notacion asinstotica (O, Θ, o, Ω,etc) siempre que se pueda.

$W_{blin}(n)= N$ $B_{blin}(n)= 1$

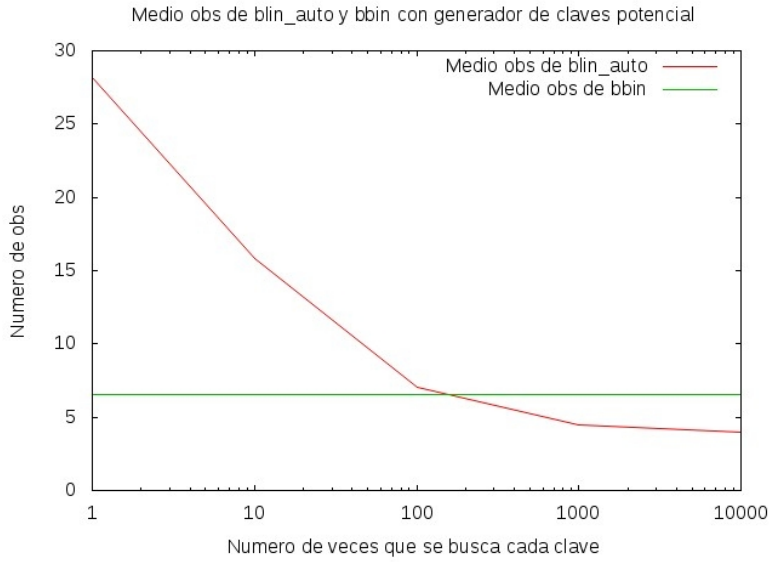
$W_{bbin}(n)=\log(N) + O(1)$ $B_{bbin}(n)=1$

5.3 Cuando se utiliza blin auto y la distribucion no uniforme dada ¿Como varia la posicion de los elementos de la lista de claves segun se van realizando mas busquedas?

Como la distribución de los números aleatorios no es uniforme, sino de forma que los números más bajos aparecen más veces, al auto organizarse la tabla hay una tendencia de encontrar antes los números más bajos al empezar la busqueda en la tabla, ya que, como aparecen más, tienden a aparecer más a la izquierda.

5.4 ¿Cual es el orden de ejecucion medio de blin auto en funcion del tamaño de elementos en el diccionario n para el caso de claves con distribucion no uniforme dado? Considerar que ya se ha realizado un elevado numero de busquedas y que la lista esta en situacion mas o menos estable.

Tras un elevado número de búsquedas es fácil ver que a partir de cierto punto, blin_auto tiene un promedio de operaciones básicas menor que bbin, y sabemos que $A_{bbin}(n)=\log(N) + O(1)$, por lo que podemos asegurar que el promedio de operaciones básicas es $A_{blin\_auto}(n)=O(\log(n))$. En la práctica podemos ver que, tras un número elevado de búsquedas, se confirma que A_{blin_auto} es menor que el de A_{bbin}.



5.5 Justifica lo mas formalmente que puedas la correccion (o dicho de otra manera, el por que busca bien) del algoritmo bbin.

El algoritmo bbin, al hacer búsqueda de coste logarítmico, es la que menos coste nos aporta, pero alcanzar este nivel de búsqueda tiene el inconveniente de que la tabla tiene que estar ordenada, es decir que en caso de que estuviese desordenanda habría que ordenar la tabla primero, pero una vez ordenada es el algoritmo más efectivo con el que hemos trabajado.

6. Conclusiones finales.

En lo que respecta a la realización de la práctica, hay varios aspectos a comentar. A la hora de comprobar en el **ejercicio1** si las funciones de búsqueda funcionaban correctamente, tuvimos problemas con bbin. El algoritmo de búsqueda binaria que habíamos implementado se basaba en recursividad, y definimos en primera instancia una función bbin y una funcion bbin_rec, llamando la primera de ellas a la segunda. Tras darnos contra la pared un par de veces (más de las que nos gustaría admitir), decidimos implementar la función de búsqueda bbin en una única función. Tras modificar algunos aspectos de la misma, funcionaba correctamente (en el código está comentado aquello que pueda dar lugar a dudas). Además, a la hora de comprobar de nuevo el funcionamiento de bbin, comprobamos que, a la hora de crear un diccionario con una tabla ordenada, los elementos no se ordenaban correctamente, por lo que era imposible ejecutar bbin de forma exitosa. Modificamos la función inserta_diccionario para que insertara los elementos correctamente (además de hacer que devolviera el número de operaciones básicas que realizaba para ello) y todo funcionó correctamente. En lo que respecta al **ejercicio2**, no tuvimos grandes problemas, a excepción de que se nos olvidó dividir el número promedio de obs entre n_veces (por ello, al observar los datos recopilados en el fichero, veíamos que no tenía sentido el resultado medio de obs respecto a max_ob y min_ob) y de que, al ver el número mínimo de obs de las búsquedas, siempre aparecía en todas 0 (el caso en el que encontraba “de primeras” la clave), por lo que modificamos sendas funciones para tener en cuenta como ob la comparación de claves donde se encontraba la clave en el diccionario.

En lo que respecta a la práctica en si, resulta bastante interesante trabajar con estos algoritmos de búsqueda pues, a pesar de sernos familiares por haberlos estudiado en otras asignaturas, a la hora de realizar gráficas y comprobar experimentalmente cómo funcionan, podemos entender mejor los mismos y tener en cuenta más cosas a la hora de ver la utilidad y pragmaticidad de un algoritmo de búsqueda. Además, resulta muy curioso y bastante útil (en ciertos casos) el algoritmo de búsqueda blin_auto, un algoritmo de búsqueda del que no teníamos constancia alguna hasta antes de esta práctica, siendo el primer algoritmo de búsqueda autorganizado que hemos vemos a día de hoy.