Análisis de Algoritmos 2017/2018

Práctica 3

Alejandro Santorum & David Cabornero, Grupo 1201.

Código	Gráficas	Memoria	Total

1. Introducción.

En esta práctica pasaremos a trabajar con algoritmos de búsqueda, trabajando con búsqueda lineal, búsqueda binaria y búsqueda lineal autoorganizada. Además, trabajaremos sobre el tema de la creación de un diccionario donde realizaremos las búsquedas y repetiremos la última parte que, como viene siendo habitual, va a consistir en medir tiempos y OB's.

2. Objetivos

2.1 Apartado 1

El primer apartado consiste básicamente en crear todas las funciones que van a afectar al diccionario o a la búsqueda dentro de él. Siendo más específicos, pretendemos crear la estructura del diccionario, una función que lo cree, otra que lo destruya, así como otras que insertan elementos.

Además, implementaremos las tres funciones que buscarán en el diccionario que hemos mencionado antes, y las aglomeraremos en un único programa que las ejecutará mediante un puntero a estas funciones.

Búsqueda lineal es una función muy simple, recorrerá la tabla desde el principio hasta el final hasta encontrar el resultado adecuado. Es la forma más simple y, por decirlo de alguna forma, la más bruta, por lo que es la menos eficiente de las tres.

Búsqueda binaria solo nos trabaja sobre tablas ordenadas, pero a cambio es mucho más eficiente que nuestra anterior función. Empieza comparando la clave a buscar con el elemento medio; si no se encuentra ahí, al estar la tabla ordenada sabremos si el elemento a encontrar puede antes o después del elemento medio. Así, ahora realizaremos recursivamente búsqueda binaria descartando una mitad completa de de la tabla, centrándonos en la mitad en la que esperamos que pueda encontrarse la clave buscada.

Búsqueda lineal autoorganizada es, evidentemente, una mejora del algoritmo de búsqueda lineal. El principio es el mismo, la búsqueda de elementos es exactamente la misma, pero al finalizar el proceso, si la clave es encontrada, se adelanta una posición el elemento, es decir, el elemento y el que le precede serán intercambiados en la tabla. Si utilizamos el generador de claves uniforme esta función no conlleva una mejora, ya que las todas las claves se generaran por igual y apenas se realizarán intercambios en la tabla. Por el contrario, si utilizamos el generador de claves potencial, el cual genera las claves de valor menor con mucha más frecuencia, está función trae consigo una gran mejora de eficiencia comparada con búsqueda lineal, ya que se posicionarán en las primeras posiciones las claves más buscadas y su búsqueda durará menos. Cabe destacar que es importante haber realizado unas cuantas búsquedas iniciales para posicionar al principio las claves más buscadas, ya que si no, en las primeras búsquedas (aunque sea con el generador de claves potencial) la función sea igual de eficiente que búsqueda lineal.

2.2 Apartado 2

Deberemos utilizar de nuevo tiempos.c, solo que no podremos reciclarlo como hicimos en la práctica 2, ya que tenemos un diccionario que es la base de nuestro nuevo programa, y que no tiene nada que ver con la forma en la que creábamos permutaciones antes, además de que la OB ahora ha cambiado. Ahora nuestra estructura ptiempo cambia, pues ahora tiene el tamaño del diccionario, el número de claves buscadas, el tiempo medio de ejecución y el máximo, mínimo y la media de veces que se ha ejecutado la OB al realizar la función de búsqueda.

Crearemos tres funciones, la primera rellena la estructura ptiempo una vez dada la función de búsqueda, la función que genera una clave a buscar aleatoria ya dada (no tenemos que hacerla), además de que debemos especificar si la tabla está ordenada, el tamaño del diccionario, el número de veces que se generará una clave a buscar y el puntero a la estructura PTIEMPO).

La segunda y tercera función son más simples. La segunda función llama a la tercera y primera función, ya que calcula los tiempos para ciertos tamaños del diccionario (desde un mínimo hasta un máximo con un cierto incremento), para lo cual hay que llamar en cada tamaño a la primera función. Después, todos los datos obtenidos se guardan en un fichero, de lo cual se encarga la tercera función.

3. Herramientas y metodología

Como siempre, hemos utilizado la página web https://c9.io/ como entorno de programación, donde utilizamos una terminal de Linux para compilar y ejecutar nuestros proyectos. Además, para detectar fugas de memoria hemos utilizado Valgrind, y para las posteriores gráficas realizadas hemos utilizado el programa recomendado GNUplot.

3.1 Apartado 1

En la ini_diccionario crearemos un diccionario vacío sabiendo el tamaño y el orden. Las comprobaciones son que el tamaño debe ser mayor que 0 (hemos visto pertinente separar casos negativos y nulos) y que el orden debe tomar el valor de la macro ORDENADO o NO_ORDENADO, obviamente. Reservaremos memoria para nuestro diccionario y comprobaremos que se ha hecho correctamente. Ídem para la tabla de la estructura diccionario. Finalmente, rellenamos la estructura: su tamaño es el tamaño pasado, su orden es el especificado y el número de datos es 0. Devolvemos el diccionario.

En libera_diccionario debemos liberar la memoria reservada en cierto diccionario. Si la tabla y el diccionario no están apuntando a NULL, se liberará primero la tabla y después el diccionario. Si la tabla es NULL, se liberará el diccionario, y si el diccionario es NULL se saldrá de la rutina sin hacer nada.

En inserta_diccionario debemos insertar un elemento dentro de la tabla de un diccionario. No miramos si la clave ya está introducida o no, ya que eso supondría una

función mucho más ineficiente. Esto también implica que, aunque haya varias soluciones, las funciones de búsqueda encontrarán solo uno de los elementos iguales. Solo hay una comprobación posible, que el diccionario que nos estén dando sea NULL. Una vez comprobado, debemos saber si nuestra tabla es una tabla ordenada o no. Si no lo es, introducimos este dato como el último y aumentamos el número de datos en un dato. Si lo es, insertamos el elemento al final y haremos swaps con el elemento anterior hasta que el elemento anterior sea menor que el elemento insertado, contando las correspondientes comparaciones de clave, que devolveremos al final del programa.

En insercion_masiva_diccionario meteremos un cierto array de claves conociendo su tamaño y el diccionario donde las insertaremos. La función notificará un error si el diccionario o el puntero a las claves es NULL o si el número de claves a introducir es negativo. Si el número de claves a introducir es 0, simplemente se finalizará la función, pues no hay claves a introducir. El programa únicamente constará de un bucle for en el que llamaremos a nuestra anterior función para introducir una a una las claves del diccionario. Devolveremos el número de comparaciones de clave totales realizadas.

En busca_diccionario, se buscará una clave en un diccionario con cierta función de búsqueda. Las comprobaciones que realizaremos serán comprobar que el puntero a función y el diccionario no sean NULL, ya que el puntero a la posición que debemos dar sí puede ser NULL, pues no importa lo que traiga, solo importa el valor que nosotros le demos. Además, debemos fiarnos del usuario para que solo nos pida realizar búsqueda lineal si la tabla está ordenada. Llamaremos a la función para que realice la búsqueda, y una vez comprobado que el puntero de la posición y ncdc no es NULL y que el número de comparaciones de clave es cero o mayor, devolveremos el ncdc. Este último error no se puede dar con nuestras funciones de búsqueda, pero lo introducimos por si se mete una función que no es nuestra. Para acabar, definiremos las tres funciones de búsqueda que se pueden pasar a esta función.

En bbin, realizaremos el algoritmo de búsqueda binaria sobre una tabla que suponemos ordenada. Comprobaremos que la tabla recibida no sea NULL y que la posición del primer elemento esté antes que la posición del último elemento o que sea la misma. Después, aplicamos el algoritmo de búsqueda binaria: visitamos el punto medio de la tabla, y si es la clave devolvemos la posición de la clave y el ncdc. Por el contrario, si no es, nos quedamos con la mitad izquierda de la tabla si la clave es menor al número medio o mayor si se da el caso contrario y repetimos el proceso hasta que encontremos la clave o hasta que la subtabla sea de un elemento y debamos concluir que la clave no está en la tabla.

En blin, aplicaremos el algoritmo de búsqueda lineal sobre una tabla que no tiene por qué estar ordenada. Comprobamos lo mismo que antes: la tabla no debe ser NULL y el primer elemento debe estar posicionado antes o en el mismo sitio que el último elemento. Ahora simplemente recorremos la tabla desde el primer elemento indicado al último indicado en busca de la clave. Se devolverán las cdc y el programa terminará en cuanto se encuentre la clave, pasando el puntero a su posición (en el caso de que se encontrara, en caso contrario el puntero será NO_ENCONTRADO).

En blin_auto realizaremos el mismo algoritmo de antes con una ligera modificación. Nos ha parecido más claro reescribir todo el código que llamar a la

función anterior, ya que así no tenemos que hacer comprobaciones sobre lo que ha devuelto la función anterior y ahorrándonos tiempo de ejecución. La función es exactamente igual que antes, pero en cuanto se encuentre una clave buscada, su posición será intercambiada con la que está un puesto antes en la lista, haciendo así que los resultados más buscados estén más próximos al inicio de la lista.

3.2 Apartado 2

En tiempo_medio_búsqueda rellenaremos la estructura PTIEMPO para cierta función de búsqueda con cierto generados de claves aleatorio, cierto orden, tamaño de diccionario y cierto número de búsquedas de clave. Las comprobaciones por lo tanto son, primero que los punteros a funciones y el puntero a la estructura PTIEMPO no apunten a NULL, además de que el orden debe estar especificado como uno de sus dos valores de la macro y que tanto el tamaño del diccionario como el número de búsquedas de clave deben ser positivos (podríamos haber contado el caso nulo como no erróneo y rellenar los valores de PTIEMPO como cero, pero no nos ha parecido que proceda tomar este caso).

Inicializaremos un diccionario, crearemos un array con una permutación, insertaremos esa permutación al diccionario, reservaremos memoria para la tabla donde se guardarán las claves a buscar y generaremos las claves correspondientes que se guardarán en la tabla anterior. En cada uno de estos cinco pasos nos aseguraremos de que no se produzca ningún error al llamar a las correspondientes funciones.

Ahora, para cada una de las claves, llamaremos a la función de búsqueda para que la busque en la tabla creada, midiendo su tiempo y en número de OB's. Si el segundo argumento nos devuelve un error o la posición del elemento buscado es NO_ENCONTRADO debemos asumir que ha habido un error.

Con esto, ya tenemos toda la estructura: el tamaño N es el tamaño del diccionario, el número de elementos a promediar es el producto del número anterior por el número de claves que se vayan a generar, el tiempo promedio y las OB medias se consigue sumando todos los tiempos u OB y dividiendo entre el número de claves generadas, y el máximo y el mínimo de OB se consiguen a base de ver cual es el mayor o menor número de OB que tiene que hacer el algoritmo en una iteración de bucle. Liberamos memoria de las tablas que contenían los datos del diccionario y las claves y también la memoria del diccionario, devolvemos OK.

En genera_tiempos_busqueda, debemos llamar al algoritmo anterior para ciertos tamaños de una tabla, comprendidos entre un mínimo y un máximo y separados por un cierto intervalo. En primer lugar, el máximo debe ser mayor que el mínimo y el incremento estrictamente positivo (si no, se genera un bucle infinito), además de que todos estos números deben ser positivos (el mínimo debe ser positivo), pues son el tamaño de una tabla. De la misma forma, debe ser positivo el número de claves generadas debe ser positivo. Además, los punteros a las dos funciones y el puntero al

fichero donde se escribirán los datos deben ser distintos de NULL y el valor de orden debe estar dentro de los valores de la macro.

Primero, calcularemos el número de tamaños distintos que vayamos a tener y reservaremos memoria para tantas estructuras PTIEMPO como tamaños distintos tengamos (con control de errores). En un bucle for, rellenaremos cada estructura PTIEMPO con su correspondiente tamaño con la función anterior y realizaremos su correspondiente control de errores. Finalmente, llamaremos a la tercera función para que imprima este array de estructuras en el fichero dado, de nuevo con su correspondiente control de errores.

La susodicha función es guarda_tabla_tiempos, donde le pasamos el fichero, el array de estructuras y el número de estructuras que contiene ese array. El puntero al fichero y a las estructuras no debe ser NULL y el número de estructuras debe ser positivo. Abrimos el fichero y comprobamos que se haya abierto correctamente (hemos decidido abrirlo de tal forma que no se sobreescriban los datos que ya se encontraran). Después, únicamente imprimimos los valores de cada estructura mediante un bucle for para, finalmente, cerrar el fichero.

4. Código fuente

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

4.1 Apartado 1

```
43  /**
44   *  Funcion: generador_claves_potencial
45   *               Esta funcion genera siguiendo una distribucion aproximadamente
46   *               potencial. Siendo los valores mas pequenos mucho mas probables
47   *               que los mas grandes. El valor 1 tiene una probabilidad del 50%,
48   *               el dos del 17%, el tres el 9%, etc.
49   */
50  void generador_claves_potencial(int *claves, int n_claves, int max){
51    int i;
52
53    for(i = 0; i < n_claves; i++) {
54      do{
55        claves[i] = (1+max) / (1 + max*((double)rand()/RAND_MAX));
56      }while(claves[i]==(1+max)); /*Se hace este cambio porque hay una posibilidad
57                                    entre RAND_MAX que rand()=0 y por lo tanto
58                                    claves[i] = max+1 lo cual no es lo que se quiere*/
59    }
60
61    return;
62  }
```

```c
/**
 *
 * Descripcion: Implementacion funciones para busqueda
 *
 * Fichero: busqueda.c
 * Autor: Carlos Aguirre
 * Version: 1.0
 * Fecha: 11-11-2016
 *
 */

#include "busqueda.h"

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <math.h>

/**
 *  Funciones de geracion de claves
 *
 *  Descripcion: Recibe el numero de claves que hay que generar
 *               en el parametro n_claves. Las claves generadas
 *               iran de 1 a max. Las claves se devuelven en
 *               el parametro claves que se debe reservar externamente
 *               a la funcion.
 */

/**
 *  Funcion: generador_claves_uniforme
 *               Esta fucnion genera todas las claves de 1 a max de forma
 *               secuencial. Si n_claves==max entonces se generan cada clave
 *               una unica vez.
 */
void generador_claves_uniforme(int *claves, int n_claves, int max){
  int i;

  for(i = 0; i < n_claves; i++) claves[i] = 1 + (i % max);

  return;
}
```

```c
/*-------------------NUESTRAS FUNCIONES----------------------*/


PDICC ini_diccionario (int tamanio, int orden){
    DICC *dic = NULL;

    /* Comprobaciones de error */
    if(tamanio < 0){
        fprintf(stdout, "Error. Tamaño del diccionario menor que cero.\n");
        return NULL;
    }

    if(tamanio == 0){
        fprintf(stdout, "Aviso. Tamaño del diccionario igual a cero.\n");
        return NULL;
    }

    if(orden != ORDENADO && orden != NO_ORDENADO){
        fprintf(stdout, "Error. Argumento orden incorrecto\n");
        return NULL;
    }
    /*-------------------------------------------------------*/
    dic = (DICC *) malloc(sizeof(DICC));
    if(dic == NULL){
        fprintf(stdout, "Error en la reserva de memoria de ini_diccionario.\n");
        return NULL;
    }

    dic->tabla = (int*) malloc (tamanio*sizeof(int));
    if(dic->tabla == NULL){
        fprintf(stdout, "Error en la reserva de memoria de la tabla de ini_diccionario.\n");
        return NULL;
    }

    dic->tamanio = tamanio;
    dic->orden = orden;
    dic->n_datos = 0;

    return dic;
}
```

```c
void libera_diccionario(PDICC pdicc){
    if(pdicc == NULL){
        fprintf(stdout, "Error. Se está intentando liberar un diccionario a NULL.\n");
        return;
    }

    if(pdicc->tabla != NULL){
        free(pdicc->tabla);
    }

    free(pdicc);
}



int inserta_diccionario(PDICC pdicc, int clave){
    int aux, j, ncdc=0;

    /* Comprobaciones de error */
    /* Suponemos que la clave que se aporta cabe en el tamaño de una variable de tipo int*/
    if(pdicc == NULL){
        fprintf(stdout,"Error. pdicc en inserta_diccionario recibido como NULL.\n");
        return ERR;
    }
    if(pdicc->tamanio == pdicc->n_datos){
        fprintf(stdout,"Error. pdicc en inserta_diccionario está lleno.\n");
        return ERR;
    }
    /*-----------------------------------------------*/
    if(pdicc->orden == NO_ORDENADO){
        pdicc->tabla[pdicc->n_datos] = clave;
        pdicc->n_datos++;
        return 0;
    }

    else if(pdicc->orden == ORDENADO){
        pdicc->tabla[pdicc->n_datos] = clave;

        aux = pdicc->tabla[pdicc->n_datos];
        j = pdicc->n_datos - 1;
        while(j >= 0 && pdicc->tabla[j] > aux){
            pdicc->tabla[j+1] = pdicc->tabla[j];
            j--;
            ncdc++;
        }
        pdicc->tabla[j+1] = aux;
        ncdc++;
        pdicc->n_datos++;
    }
    return ncdc;
}

```

```c
int insercion_masiva_diccionario (PDICC pdicc, int *claves, int n_claves){
    int i, ncdc=0, aux=0;

    /* Comprobaciones */
    if(pdicc == NULL){
        fprintf(stdout,"Error. pdicc en inserta_masiva_diccionario recibido como NULL.\n");
        return ERR;
    }

    if(claves == NULL){
        fprintf(stdout,"Error. claves en insercion_masiva_diccionario recibido como NULL.\n");
        return ERR;
    }

    if(n_claves <= 0){
        fprintf(stdout,"Error. n_claves en insercion_masiva_diccionario negativo o nulo.\n");
        return ERR;
    }

    if(n_claves == 0){
        return 0;
    }
    /*-------------------------------------------------------------------------*/

    for(i=0;i<n_claves;i++){
        aux = inserta_diccionario(pdicc, claves[i]);
        if(aux == ERR){
            fprintf(stdout,"Error. retorno de inserta_diccionario en insercion_masiva erróneo.\n");
            return ERR;
        }
        ncdc += aux;
    }

    return ncdc;
}
```

```c
int busca_diccionario(PDICC pdicc, int clave, int *ppos, pfunc_busqueda metodo){
    int ncdc=0;
    /* Comprobaciones de error */
    if(pdicc == NULL){
        fprintf(stdout,"Error. pdicc en busca_diccionario recibido como NULL.\n");
        return ERR;
    }
    /* OBSERVACIÓN: No se necesita una comprobación para ppos ya que este
       puede que sea igual a NULL, dependiendo de como se pase en el programa principal.*/
    if(metodo == NULL){
        fprintf(stdout,"Error. metodo en busca_diccionario recibido como NULL.\n");
        return ERR;
    }
    /* ------------------------------------------------------- */

    ncdc = metodo(pdicc->tabla, 0, pdicc->n_datos-1, clave, ppos);
    if(ncdc == ERR || ncdc < 0){ /* ERROR */
        fprintf(stdout, "Error en la función de búsqueda en busca_diccionario.\n");
        return ERR;
    }
    if(ppos == NULL){ /* ERROR */
        fprintf(stdout, "ppos es igual a NULL despues de una función de búsqueda, lo cual es imposible.\n");
        return ERR;
    }

    return ncdc;
}
```

```c
/* Funciones de busqueda del TAD Diccionario */
int bbin(int *tabla,int P,int U,int clave,int *ppos){
    int m, ncdc=0;

    /* Comprobaciones de error*/
    if(tabla == NULL){
        fprintf(stdout,"Error. tabla en bbin recibida como NULL.\n");
        return ERR;
    }
    if(P > U){
        fprintf(stdout,"Error. primero mayor que último en bbin.\n");
        return ERR;
    }
    /*-----------------------------------------------------*/
    while(P <= U){
        m = (P+U)/2;

        ncdc++;
        if(tabla[m] == clave){
            *ppos = m;
            return ncdc;
        }
        else if(clave < tabla[m]){
            U = m-1;
        }

        else{
            P = m+1;
        }
    }

    *ppos = NO_ENCONTRADO;

    return ncdc;
}
```

```c
int blin(int *tabla,int P,int U,int clave,int *ppos){
    int ncdc=0, i;

    /* Comprobaciones de error*/
    if(tabla == NULL){
        fprintf(stdout,"Error. tabla en bbin recibida como NULL.\n");
        return ERR;
    }
    if(P > U){
        fprintf(stdout,"Error. primero mayor que último en bbin.\n");
        return ERR;
    }
    /*------------------------------------------------*/
    for(i=P; i<=U; i++){
        ncdc++;
        if(tabla[i] == clave){
            *ppos = i;
            return ncdc;
        }
    }
    *ppos = NO_ENCONTRADO;

    return ncdc;
}
```

```c
int blin_auto(int *tabla,int P,int U,int clave,int *ppos){
    int ncdc=0, i, aux;
    /*Podríamos haber utilizado el algoritmo anterior blin en este código, pero nos parece
    más claro en lo que respecta a comprobaciones volver a representar todo el algoritmo aquí*/

    /* Comprobaciones de error*/
    if(tabla == NULL){
        fprintf(stdout,"Error. tabla en bbin recibida como NULL.\n");
        return ERR;
    }
    if(P > U){
        fprintf(stdout,"Error. primero mayor que último en bbin.\n");
        return ERR;
    }
    /*------------------------------------------------*/

    for(i=P; i<=U; i++){
        ncdc++;
        if(tabla[i] == clave){
            *ppos = i;
            if(i != P){
                aux = tabla[i]; /*Swap*/
                tabla[i] = tabla[i-1];
                tabla[i-1] = aux;
            }
            return ncdc;
        }
    }
    *ppos = NO_ENCONTRADO;

    return ncdc;
}
```

```c
/**
 *
 * Descripcion: Implementacion de funciones de generacion de permutaciones
 *
 * Fichero: permutaciones.c
 * Autor: Carlos Aguirre
 * Version: 1.0
 * Fecha: 16-09-2017
 *
 */


#include "permutaciones.h"

/***************************************************/
/* Funcion: aleat_num Fecha: 22/09/2017            */
/* Autores: David Cabornero Pascual                */
/*          Alejandro Santorum Varela              */
/* Rutina que genera un numero aleatorio           */
/* entre dos numeros dados                         */
/*                                                 */
/* Entrada:                                        */
/* int inf: limite inferior                        */
/* int sup: limite superior                        */
/* Salida:                                         */
/* int: numero aleatorio                           */
/***************************************************/
int aleat_num(int inf, int sup){
    int result = 0;

    if(inf == sup){
        return sup;
    }

    else if (inf > sup){
        printf("ERROR: Límite inferior mayor que el límite superior.\n");
        exit(-1);
    }

    result = (inf + ((int) ((((double)(sup-inf+1)) * rand())/(RAND_MAX + 1.0))));

    return result;
}

/***************************************************/
/* Funcion: genera_perm Fecha: 22/09/2017          */
/* Autores: David Cabornero Pascual                */
/*          Alejandro Santorum Varela              */
/* Rutina que genera una permutacion               */
/* aleatoria                                       */
/*                                                 */
/* Entrada:                                        */
/* int n: Numero de elementos de la                */
/* permutacion                                     */
/* Salida:                                         */
/* int *: puntero a un array de enteros            */
/* que contiene a la permutacion                   */
/* o NULL en caso de error                         */
/***************************************************/
int* genera_perm(int N){
    int i, aux1, aux2;
    int *perm;

    if(N<=0){
        printf("ERROR: Tamanio del array de la permutación menor o igual a cero.\n");
        return NULL;
    }
    else if(N>MAX_TAM){
        printf("MEDIDA DE SEGURIDAD: Tamanio del array de la permutación excesivo (superior a INT_MAX).\n");
        return NULL;
    }

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

    for(i=0; i<N; i++){
        perm[i] = i+1;
    }

    for(i=0; i<N; i++){
        aux1 = perm[i];
        aux2 = aleat_num(i, N-1);
        perm[i] = perm[aux2];
        perm[aux2] = aux1;
    }

    return perm;
}
```

```c
/***************************************************/
/* Funcion: genera_permutaciones Fecha: 22/09/2017 */
/* Autores: David Cabornero Pascual                */
/*          Alejandro Santorum Varela              */
/*                                                 */
/* Funcion que genera n_perms permutaciones        */
/* aleatorias de tamanio elementos                 */
/*                                                 */
/* Entrada:                                        */
/* int n_perms: Numero de permutaciones            */
/* int N: Numero de elementos de cada              */
/* permutacion                                     */
/* Salida:                                         */
/* int**: Array de punteros a enteros              */
/* que apuntan a cada una de las                   */
/* permutaciones                                   */
/* NULL en caso de error                           */
/***************************************************/
int** genera_permutaciones(int n_perms, int N){
    int i,j;
    int **pp=NULL;

    if(N<=0 || n_perms<=0){
        printf("ERROR: El número de permutaciones y/o el tamanio de las mismas menor o igual que cero.\n");
        return NULL;
    }
    else if(N>MAX_TAM||n_perms>MAX_TAM){
        printf("MEDIDA DE SEGURIDAD: El número de permutaciones y/o el tamanio de las mismas es demasiado grande.\n");
    }

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

    for(i=0; i<n_perms; i++){
        pp[i] = genera_perm(N);
        if(pp[i] == NULL){
            for(j=0; j<i; j++){
                free(pp[j]);
            }
            free(pp);
            return NULL;
        }
    }
    return pp;
}
```

```c
/***************************************************/
/* Programa: ejercicio1     Fecha: 13/12/2017      */
/* Autores: Alejandro Santorum                     */
/*          David Cabornero                        */
/*                                                 */
/* Programa que comprueba el funcionamiento de     */
/* la busqueda lineal                              */
/*                                                 */
/* Entrada: Linea de comandos                      */
/*   -tamanio: numero elementos diccionario        */
/*   -clave:   clave a buscar                      */
/*                                                 */
/* Salida: 0: OK, -1: ERR                          */
/***************************************************/

#include<stdlib.h>
#include<stdio.h>
#include<string.h>
#include<time.h>

#include "permutaciones.h"
#include "busqueda.h"

int main(int argc, char** argv)
{
  int i, nob, pos;
  unsigned int clave, tamanio;
  PDICC pdicc;
  int *perm;

  srand(time(NULL));

  if (argc != 5) {
    fprintf(stderr, "Error en los parametros de entrada:\n\n");
    fprintf(stderr, "%s -tamanio <int> -clave <int>\n", argv[0]);
    fprintf(stderr, "Donde:\n");
    fprintf(stderr, " -tamanio : numero elementos de la tabla.\n");
    fprintf(stderr, " -clave : clave a buscar.\n");
    exit(-1);
  }

  printf("Practica numero 3, apartado 1\n");
  printf("Realizada por: Alejandro Santorum y David Cabornero.\n");
  printf("Grupo: 1201\n");
  printf("Pareja 10.\n");
```

```c
  /* comprueba la linea de comandos */
  for(i = 1; i < argc; i++) {
    if (strcmp(argv[i], "-tamanio") == 0) {
      tamanio = atoi(argv[++i]);
    } else if (strcmp(argv[i], "-clave") == 0) {
      clave = atoi(argv[++i]);
    } else {
      fprintf(stderr, "Parametro %s es incorrecto\n", argv[i]);
    }
  }

  pdicc = ini_diccionario(tamanio, NO_ORDENADO);
  if (pdicc == NULL) {
    /* error */
    printf("Error: No se puede Iniciar el diccionario\n");
    exit(-1);
  }

  perm = genera_perm(tamanio);
  if (perm == NULL) {
    /* error */
    printf("Error: No hay memoria\n");
    libera_diccionario(pdicc);
    exit(-1);
  }

  nob = insercion_masiva_diccionario(pdicc, perm, tamanio);
  if (nob == ERR) {
    /* error */
    printf("Error: No se puede crear el diccionario memoria\n");
    free(perm);
    libera_diccionario(pdicc);
    exit(-1);
  }

```

```c
  nob = busca_diccionario(pdicc,clave,&pos,blin_auto);


  if(nob == ERR){ /* Caso de error */
    printf("Error al buscar la clave %d\n", clave);
  }
  else if(pos == NO_ENCONTRADO){ /* CLave no encontrada */
    printf("La clave %d no se encontro en la tabla\n",clave);
  }
  else{ /* Caso en el que la clave se ha encontrado */
    printf("Clave %d encontrada en la posicion %d en %d op. basicas\n", clave, pos, nob);
  }

/*
  if(nob >= 0) {
    printf("Clave %d encontrada en la posicion %d en %d op. basicas\n", clave, pos, nob);
  } else if (nob==NO_ENCONTRADO) {
    printf("La clave %d no se encontro en la tabla\n",clave);
  } else {
    printf("Error al buscar la clave %d\n",clave);
  }
*/

  free(perm);
  libera_diccionario(pdicc);

  return 0;
}

```

4.2 Apartado 2

```c
/**
 *
 * Descripcion: Implementacion de funciones de tiempo
 *
 * Fichero: tiempos.c
 * Autor: Carlos Aguirre Maeso
 * Version: 1.0
 * Fecha: 16-09-2017
 *
 */

#include "tiempos.h"
#include "busqueda.h"
#include "permutaciones.h"
#include <limits.h>
#include <time.h>
#include <math.h>

#define TENTOTHENINE 1000000000
#define TENTOTHESIX 1000000

/***************************************************/
/* Funcion: tiempo_medio_ordenación     Fecha: 20/10/2017 */
/* Autores: David Cabornero Pascual                */
/*          Alejandro Santorum Varela              */
/*                                                 */
/* Entrada:                                        */
/* int n_perms: Numero de permutaciones            */
/* int N: Numero de elementos de cada              */
/* ptiempo: puntero a estructura Tiempo            */
/* permutacion                                     */
/* metodo: puntero a función de ordenación         */
/* Salida:                                         */
/* short: OK en caso de éxito, ERR en caso de ERROR */
/***************************************************/
short tiempo_medio_busqueda(pfunc_busqueda metodo, pfunc_generador_claves generador, int orden, int N, int n_veces, PTIEMPO ptiempo){
  DICC *dicc = NULL;
  int *perm=NULL, *tabla_claves=NULL;
  int pos, clave, check1, check2;
  long i, nob, max=0, min=INT_MAX, n_claves;
  double media = 0, actual, total=0;
  struct timespec start, end, aux;
```

```c
  /*Comprobaciones de error-----------------------------------------------------*/
  if(metodo == NULL){
    printf("Error. Puntero a función nulo (metodo).\n");
    return ERR;
  }
  if(generador == NULL){
    printf("Error. Puntero a función nulo (generador).\n");
    return ERR;
  }
  if(ptiempo == NULL){
    printf("Error. Puntero a la estructura tiempo nulo.\n");
    return ERR;
  }
  if(n_veces<=0){
    printf("Error. Número de veces que se busca cada una de las claves nulo o negativo.\n");
    return ERR;
  }
  if(N<=0){
    printf("Error. Tamaño de las permutaciones negativo o nulo.\n");
    return ERR;
  }
  if(orden != ORDENADO && orden != NO_ORDENADO){
    printf("Error. Orden no especificado.\n");
    return ERR;
  }

  /*El grueso del programa------------------------------------------------------*/
  n_claves = N*n_veces;
  ptiempo->N = N;
  ptiempo->n_elems = n_claves;

  dicc = ini_diccionario(N, orden);
  if(dicc == NULL){
    printf("Error al crear un diccionario\n");
    return ERR;
  }

  perm = genera_perm(N);
  if (perm == NULL) {
    /* error */
    printf("Error: No hay memoria en perm.\n");
    libera_diccionario(dicc);
    return ERR;
  }
```

```c
  nob = insercion_masiva_diccionario(dicc, perm, N);
  if (nob == ERR) {
    /* error */
    printf("Error al llamar a insercion_masiva_diccionario\n");
    free(perm);
    libera_diccionario(dicc);
    return ERR;
  }

  tabla_claves = (int *) malloc(n_claves*sizeof(int));
  if(tabla_claves==NULL){
    /* error */
    printf("Error al llamar a insercion_masiva_diccionario\n");
    free(perm);
    libera_diccionario(dicc);
    return ERR;
  }

  generador(tabla_claves, n_claves, N);
  if(tabla_claves == NULL){
    printf("Error al generar claves\n");
    free(perm);
    libera_diccionario(dicc);
    return ERR;
  }

  for(i=0; i<n_claves; i++){
    clave = tabla_claves[i];
    check1 = clock_gettime(CLOCK_REALTIME, &start);
    actual = busca_diccionario(dicc, clave, &pos, metodo);
    check2 = clock_gettime(CLOCK_REALTIME, &end);

    if(check1 == -1){
      printf("Error en clock_gettime (start) en tiempo_medio_busqueda.\n");
      libera_diccionario(dicc);
      free(tabla_claves);
      free(perm);
      return ERR;
    }
    if(check2 == -1){
      printf("Error en clock_gettime (end) en tiempo_medio_busqueda.\n");
      libera_diccionario(dicc);
      free(tabla_claves);
      free(perm);
      return ERR;
    }
```

```c
    if(actual == ERR){
      printf("Error en busca_diccionario en tiempo_medio_busqueda (ERR).\n");
      libera_diccionario(dicc);
      free(tabla_claves);
      free(perm);
      return ERR;
    }
    if(pos == NO_ENCONTRADO){
      printf("Error en busca_diccionario en tiempo_medio_busqueda (NO_ENCONTRADO).\n");
      libera_diccionario(dicc);
      free(tabla_claves);
      free(perm);
      return ERR;
    }

    aux.tv_sec = end.tv_sec - start.tv_sec; /* Calculamos los segundos */
    aux.tv_nsec = end.tv_nsec - start.tv_nsec; /* Calculamos los nanosegundos */
    total = total + aux.tv_sec*TENTOTHENINE + aux.tv_nsec; /* Guardamos el tiempo en nanosegundos */

    media = media + actual;
    if(actual > max){
      max = actual;
    }
    if(actual < min){
      min = actual;
    }
  }
  total = total/n_claves; /* Recordemos que n_claves = n_elems = N * n_veces */

  media = media/n_claves;

  ptiempo->medio_ob = media;
  ptiempo->max_ob = max;
  ptiempo->min_ob = min;
  ptiempo->tiempo = total/TENTOTHENINE; /*Guardamos el tiempo en la struct en segundos
                                          tal y como se indica en la documentación */

  free(tabla_claves);
  free(perm);
  libera_diccionario(dicc);

  return OK;
}
```

```c
/*****************************************************/
/* Funcion: genera_tiempos_busqueda    Fecha: 20/10/2017*/
/* Autores: David Cabornero Pascual                     */
/*          Alejandro Santorum Varela                   */
/*                                                      */
/* Entrada:                                             */
/* int n_perms: Numero de permutaciones                 */
/* int num_min: tamaño mínimo permutación               */
/* int num_max: tamaño máximo de permutación            */
/* permutacion                                          */
/* int incr: incremento del tamaño.                     */
/* metodo: puntero a función de ordenación              */
/* fichero: puntero a un fichero donde se imprimirán datos*/
/* Salida:                                              */
/* short: OK en caso de éxito, ERR en caso de ERROR     */
/*****************************************************/
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){
  TIEMPO *ptiempo;
  int array_size, i, j;
  double aux;

  /*Comprobaciones*/
  if(num_min<0 || num_max < num_min || incr <= 0){
    printf("Error. Meta unos números decentes por favor.\n");
    return ERR;
  }
  if(n_veces<=0){
    printf("Error. Número de permutaciones nulo o negativo.\n");
    return ERR;
  }
  if(orden != ORDENADO && orden != NO_ORDENADO){
    printf("Error. Orden no especificado.\n");
    return ERR;
  }
  if(fichero == NULL){
    printf("Error. Puntero a fichero nulo.\n");
    return ERR;
  }
  if(metodo == NULL){
    printf("Error. Puntero a función nulo (metodo).\n");
    return ERR;
  }
  if(generador == NULL){
    printf("Error. Puntero a función nulo (generador).\n");
    return ERR;
  }
  /*-------------------------------------------*/
```

51:12 C and C++

```c
  /*Grueso del programa*/
  aux = num_max - num_min;
  aux = aux/incr; /* ya nos hemos asegurado anteriormente que incr != 0 */
  array_size = ceil(aux);
  printf("array = %d\n", array_size);
  /*El anterior printf, intuitivamente, te dice el número
  de filas que se van a imprimir en el fichero de salida*/

  ptiempo = (TIEMPO *) malloc(array_size * sizeof(TIEMPO));
  if(ptiempo == NULL){
    printf("Error. Reserva de memoria en PTIEMPO.\n");
    return ERR;
  }

  for(i=num_min, j=0; j<array_size; i = i + incr, j++){
    if(tiempo_medio_busqueda(metodo, generador, orden, i, n_veces, &ptiempo[j]) == ERR){
      printf("Error en la función 2 haciendo la 1. del ej2.\n");
      return ERR;
    }
  }

  if(guarda_tabla_tiempos(fichero, ptiempo, array_size) == ERR){
    free(ptiempo);
    return ERR;
  }

  free(ptiempo);

  return OK;
}
```

```c
/* Funcion: guarda_tabla_tiempos         Fecha: 20/10/2017*/
/* Autores: David Cabornero Pascual                      */
/*          Alejandro Santorum Varela                    */
/* Entrada:                                              */
/* int n_tiempos: tamaño del array                       */
/* fichero: puntero a un fichero donde se imprimirán datos*/
/* ptiempo: array de punteros a estructura tiempo        */
/*                                                       */
/* Salida:                                               */
/* short: OK en caso de éxito, ERR en caso de ERROR      */
/*******************************************************/
short guarda_tabla_tiempos(char *fichero, TIEMPO *ptiempo, int n_tiempos){
  FILE *f=NULL;
  int i;
  /*Comprobaciones*/
  if(fichero == NULL){
    printf("Error. Puntero a fichero nulo.\n");
    return ERR;
  }
  if(ptiempo == NULL){
    printf("Error. Puntero a la estructura tiempo nulo.\n");
    return ERR;
  }
  if(n_tiempos <= 0){
    printf("Error. Tamaño del array tiempo negativo o nulo.\n");
    return ERR;
  }
  /*Grueso del programa*/
  f = (FILE *) fopen(fichero, "a");
  if(f == NULL){
    return ERR;
  }
  fprintf(f, "size   time(us)   avg_ob   max_ob   min_ob.\n");

  for(i=0; i<n_tiempos; i++){
    fprintf(f, "%d   %f   %.2f   %d   %d\n", ptiempo[i].N, ptiempo[i].tiempo * TENTOTHESIX, ptiempo[i].medio_ob, ptiempo[i].max_ob, ptiempo[i].min_ob);
  } /* Multiplicamos el tiempo por 10^6 para pasar de segundos a microsegundos. Así su interpretación es más sencilla en la gráfica */
  fclose(f);
  return OK;
}
```

308:1 C and C++ Spaces: 2

```c
/*******************************************************/
/* Programa: ejercicio2 Fecha: 13/12/2017              */
/* Autores:  Alejandro Santorum & David Cabornero      */
/*                                                     */
/* Programa que escribe en un fichero                  */
/* los tiempos medios del algoritmo de                 */
/* busqueda                                            */
/*                                                     */
/* Entrada: Linea de comandos                          */
/* -num_min: numero minimo de elementos de la tabla    */
/* -num_max: numero minimo de elementos de la tabla    */
/* -incr: incremento                                   */
/* -fclaves: numero de claves a buscar                 */
/* -numP: Introduce el numero de permutaciones a promediar */
/* -fichSalida: Nombre del fichero de salida           */
/*                                                     */
/* Salida: 0 si hubo error                             */
/*         -1 en caso contrario                        */
/*******************************************************/

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <time.h>
#include "permutaciones.h"
#include "busqueda.h"
#include "tiempos.h"

int main(int argc, char** argv)
{
  int i, num_min,num_max,incr,n_veces;
  char nombre[256];
  short ret;

  srand(time(NULL));

  if (argc != 11) {
    fprintf(stderr, "Error en los parametros de entrada:\n\n");
    fprintf(stderr, "%s -num_min <int> -num_max <int> -incr <int> -n_veces <int> -fichSalida <string>\n", argv[0]);
    fprintf(stderr, "Donde:\n");
    fprintf(stderr, "-num_min: numero minimo de elementos de la tabla\n");
    fprintf(stderr, "-num_max: numero minimo de elementos de la tabla\n");
    fprintf(stderr, "-incr: incremento\n");
    fprintf(stderr, "-n_veces: numero de veces que se busca cada clave\n");
    fprintf(stderr, "-fichSalida: Nombre del fichero de salida\n");
    exit(-1);
  }
```

```c
  printf("Practica numero 3, apartado 2\n");
  printf("Realizada por: Alejandro Santorum y David Cabornero.\n");
  printf("Grupo: 1201.\n");
  printf("pareja 10.\n");

  /* comprueba la linea de comandos */
  for(i = 1; i < argc ; i++) {
    if (strcmp(argv[i], "-num_min") == 0) {
      num_min = atoi(argv[++i]);
    } else if (strcmp(argv[i], "-num_max") == 0) {
      num_max = atoi(argv[++i]);
    } else if (strcmp(argv[i], "-incr") == 0) {
      incr = atoi(argv[++i]);
    } else if (strcmp(argv[i], "-n_veces") == 0) {
      n_veces = atoi(argv[++i]);
    } else if (strcmp(argv[i], "-fichSalida") == 0) {
      strcpy(nombre, argv[++i]);
    } else {
      fprintf(stderr, "Parametro %s es incorrecto\n", argv[i]);
      exit(-1);
    }
  }

  /* calculamos los tiempos */
  ret = genera_tiempos_busqueda(blin_auto, generador_claves_potencial, NO_ORDENADO, nombre, num_min, num_max, incr, n_veces);
  if (ret == ERR) {
    printf("Error en la funcion genera_tiempos_busqueda\n");
    exit(-1);
  }

  printf("Salida correcta \n");

  return 0;
}
```

5. Resultados, Gráficas

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

5.1 Apartado 1

Salida búsqueda lineal clave encontrada (I):

```
santorum:~/workspace/practice3 (master) $ valgrind --leak-check=full ./ejercicio1 -tamanio 100 -clave 50
==1171== Memcheck, a memory error detector
==1171== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==1171== Using Valgrind-3.10.1 and LibVEX; rerun with -h for copyright info
==1171== Command: ./ejercicio1 -tamanio 100 -clave 50
==1171==
Practica numero 3, apartado 1
Realizada por: Alejandro Santorum y David Cabornero.
Grupo: 1201
Pareja 10.
Clave 50 encontrada en la posicion 17 en 18 op. basicas
==1171==
==1171== HEAP SUMMARY:
==1171==     in use at exit: 0 bytes in 0 blocks
==1171==   total heap usage: 3 allocs, 3 frees, 824 bytes allocated
==1171==
==1171== All heap blocks were freed -- no leaks are possible
==1171==
==1171== For counts of detected and suppressed errors, rerun with: -v
==1171== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Salida búsqueda lineal clave encontrada (II):

```
santorum:~/workspace/practice3 (master) $ valgrind --leak-check=full ./ejercicio1 -tamanio 100 -clave 100
==1176== Memcheck, a memory error detector
==1176== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==1176== Using Valgrind-3.10.1 and LibVEX; rerun with -h for copyright info
==1176== Command: ./ejercicio1 -tamanio 100 -clave 100
==1176==
Practica numero 3, apartado 1
Realizada por: Alejandro Santorum y David Cabornero.
Grupo: 1201
Pareja 10.
Clave 100 encontrada en la posicion 53 en 54 op. basicas
==1176==
==1176== HEAP SUMMARY:
==1176==     in use at exit: 0 bytes in 0 blocks
==1176==   total heap usage: 3 allocs, 3 frees, 824 bytes allocated
==1176==
==1176== All heap blocks were freed -- no leaks are possible
==1176==
==1176== For counts of detected and suppressed errors, rerun with: -v
==1176== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Salida búsqueda lineal clave no encontrada:

```
santorum:~/workspace/practice3 (master) $ valgrind --leak-check=full ./ejercicio1 -tamanio 100 -clave 101
==1187== Memcheck, a memory error detector
==1187== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==1187== Using Valgrind-3.10.1 and LibVEX; rerun with -h for copyright info
==1187== Command: ./ejercicio1 -tamanio 100 -clave 101
==1187==
Practica numero 3, apartado 1
Realizada por: Alejandro Santorum y David Cabornero.
Grupo: 1201
Pareja 10.
La clave 101 no se encontro en la tabla
==1187==
==1187== HEAP SUMMARY:
==1187==     in use at exit: 0 bytes in 0 blocks
==1187==   total heap usage: 3 allocs, 3 frees, 824 bytes allocated
==1187==
==1187== All heap blocks were freed -- no leaks are possible
==1187==
==1187== For counts of detected and suppressed errors, rerun with: -v
==1187== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Salida búsqueda binaria clave encontrada (I):

```
santorum:~/workspace/practice3 (master) $ valgrind --leak-check=full ./ejercicio1 -tamanio 100 -clave 50
==1210== Memcheck, a memory error detector
==1210== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==1210== Using Valgrind-3.10.1 and LibVEX; rerun with -h for copyright info
==1210== Command: ./ejercicio1 -tamanio 100 -clave 50
==1210==
Practica numero 3, apartado 1
Realizada por: Alejandro Santorum y David Cabornero.
Grupo: 1201
Pareja 10.
Clave 50 encontrada en la posicion 49 en 7 op. basicas
==1210==
==1210== HEAP SUMMARY:
==1210==     in use at exit: 0 bytes in 0 blocks
==1210==   total heap usage: 3 allocs, 3 frees, 824 bytes allocated
==1210==
==1210== All heap blocks were freed -- no leaks are possible
==1210==
==1210== For counts of detected and suppressed errors, rerun with: -v
==1210== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Salida búsqueda binaria clave encontrada (II):

```
santorum:~/workspace/practice3 (master) $ valgrind --leak-check=full ./ejercicio1 -tamanio 100 -clave 100
==1215== Memcheck, a memory error detector
==1215== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==1215== Using Valgrind-3.10.1 and LibVEX; rerun with -h for copyright info
==1215== Command: ./ejercicio1 -tamanio 100 -clave 100
==1215==
Practica numero 3, apartado 1
Realizada por: Alejandro Santorum y David Cabornero.
Grupo: 1201
Pareja 10.
Clave 100 encontrada en la posicion 99 en 6 op. basicas
==1215==
==1215== HEAP SUMMARY:
==1215==     in use at exit: 0 bytes in 0 blocks
==1215==   total heap usage: 3 allocs, 3 frees, 824 bytes allocated
==1215==
==1215== All heap blocks were freed -- no leaks are possible
==1215==
==1215== For counts of detected and suppressed errors, rerun with: -v
==1215== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Salida búsqueda binaria clave no encontrada:

```
santorum:~/workspace/practice3 (master) $ valgrind --leak-check=full ./ejercicio1 -tamanio 100 -clave 101
==1545== Memcheck, a memory error detector
==1545== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==1545== Using Valgrind-3.10.1 and LibVEX; rerun with -h for copyright info
==1545== Command: ./ejercicio1 -tamanio 100 -clave 101
==1545==
Practica numero 3, apartado 1
Realizada por: Alejandro Santorum y David Cabornero.
Grupo: 1201
Pareja 10.
La clave 101 no se encontro en la tabla
==1545==
==1545== HEAP SUMMARY:
==1545==     in use at exit: 0 bytes in 0 blocks
==1545==   total heap usage: 3 allocs, 3 frees, 824 bytes allocated
==1545==
==1545== All heap blocks were freed -- no leaks are possible
==1545==
==1545== For counts of detected and suppressed errors, rerun with: -v
==1545== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Salida búsqueda lineal autoorganizada clave encontrada (I):

```
santorum:~/workspace/practice3 (master) $ valgrind --leak-check=full ./ejercicio1 -tamanio 100 -clave 50
==1567== Memcheck, a memory error detector
==1567== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==1567== Using Valgrind-3.10.1 and LibVEX; rerun with -h for copyright info
==1567== Command: ./ejercicio1 -tamanio 100 -clave 50
==1567==
Practica numero 3, apartado 1
Realizada por: Alejandro Santorum y David Cabornero.
Grupo: 1201
Pareja 10.
Clave 50 encontrada en la posicion 50 en 51 op. basicas
==1567==
==1567== HEAP SUMMARY:
==1567==     in use at exit: 0 bytes in 0 blocks
==1567==   total heap usage: 3 allocs, 3 frees, 824 bytes allocated
==1567==
==1567== All heap blocks were freed -- no leaks are possible
==1567==
==1567== For counts of detected and suppressed errors, rerun with: -v
==1567== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Salida búsqueda lineal autoorganizada clave encontrada (II):

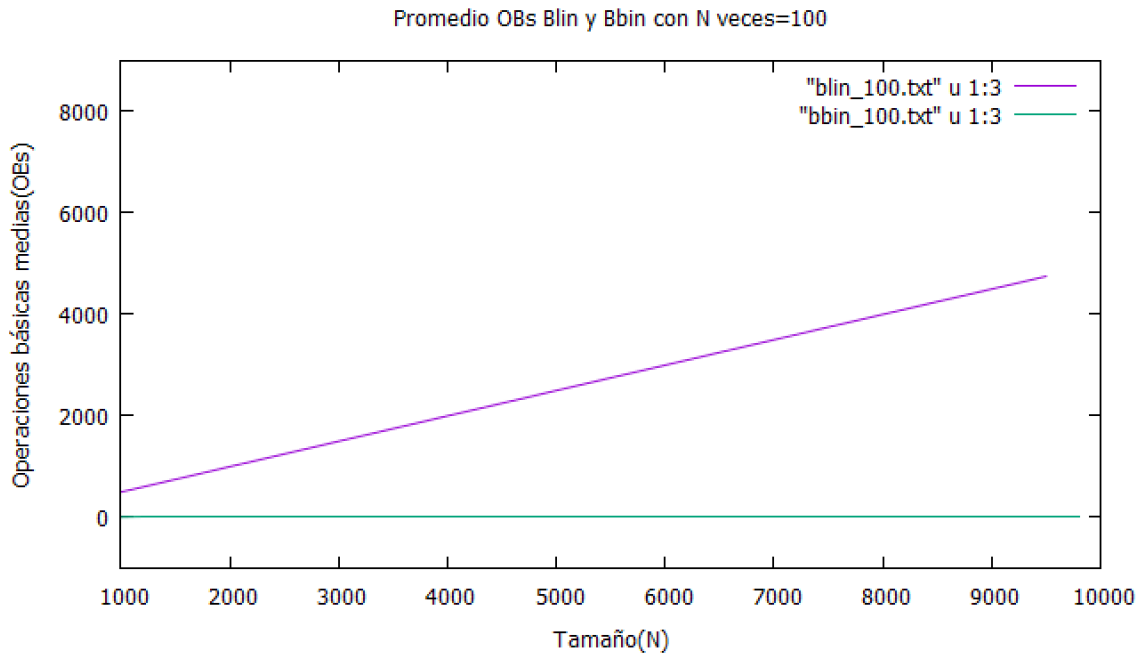
```
santorum:~/workspace/practice3 (master) $ valgrind --leak-check=full ./ejercicio1 -tamanio 100 -clave 100
==1578== Memcheck, a memory error detector
==1578== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==1578== Using Valgrind-3.10.1 and LibVEX; rerun with -h for copyright info
==1578== Command: ./ejercicio1 -tamanio 100 -clave 100
==1578==
Practica numero 3, apartado 1
Realizada por: Alejandro Santorum y David Cabornero.
Grupo: 1201
Pareja 10.
Clave 100 encontrada en la posicion 65 en 66 op. basicas
==1578==
==1578== HEAP SUMMARY:
==1578==     in use at exit: 0 bytes in 0 blocks
==1578==   total heap usage: 3 allocs, 3 frees, 824 bytes allocated
==1578==
==1578== All heap blocks were freed -- no leaks are possible
==1578==
==1578== For counts of detected and suppressed errors, rerun with: -v
==1578== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Salida búsqueda lineal autoorganizada clave no encontrada:

```
santorum:~/workspace/practice3 (master) $ valgrind --leak-check=full ./ejercicio1 -tamanio 100 -clave 101
==1583== Memcheck, a memory error detector
==1583== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==1583== Using Valgrind-3.10.1 and LibVEX; rerun with -h for copyright info
==1583== Command: ./ejercicio1 -tamanio 100 -clave 101
==1583==
Practica numero 3, apartado 1
Realizada por: Alejandro Santorum y David Cabornero.
Grupo: 1201
Pareja 10.
La clave 101 no se encontro en la tabla
==1583==
==1583== HEAP SUMMARY:
==1583==     in use at exit: 0 bytes in 0 blocks
==1583==   total heap usage: 3 allocs, 3 frees, 824 bytes allocated
==1583==
==1583== All heap blocks were freed -- no leaks are possible
==1583==
==1583== For counts of detected and suppressed errors, rerun with: -v
==1583== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

5.2 Apartado 2

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



Comentarios:

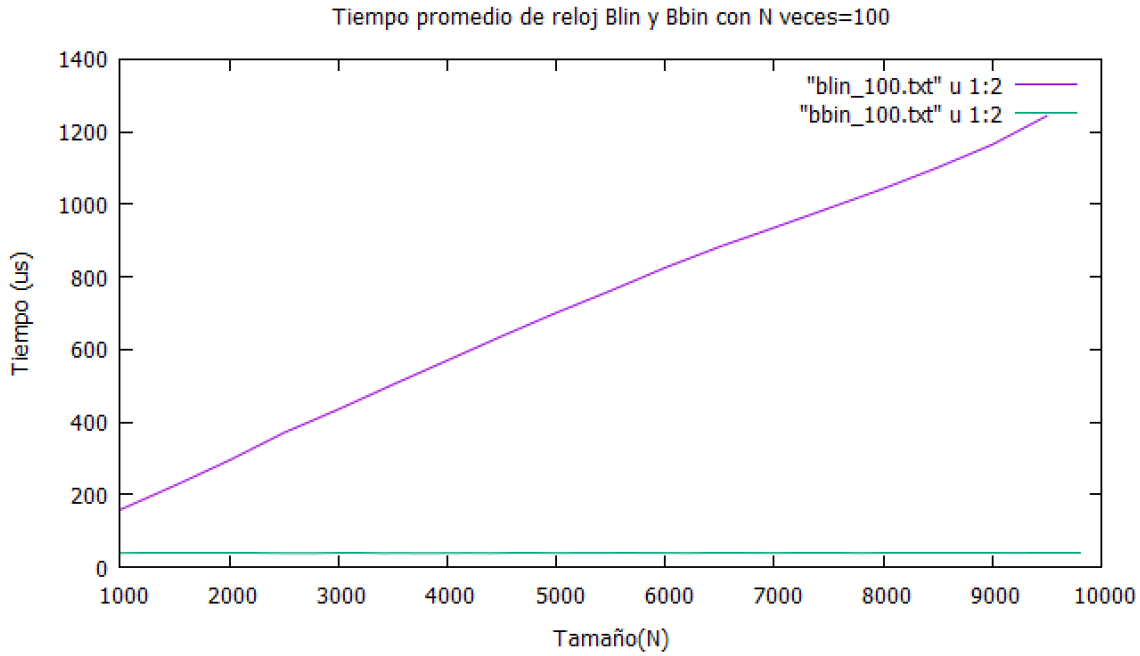
Se muestra el número promedio de OB´s de búsqueda lineal (línea superior) y de búsqueda binaria (línea inferior).

Da la sensación que para Bbin el número de OB's es cero, pero eso es debido a la escala. En realizadad va desde 2 a 12, lo cual no es apreciable al tener que representar

la gráfica de Blin, la cual sigue estrictamente el mismo comportamiento que la función N/2 (O(N)), lo que es lógico ya que algunas claves las encontrará al final de la tabla necesitando cerca de N comparaciones, y otras al principio utilizando muy pocas comparaciones, lo que origina el promedio de N/2 OB's.

Por el otro lado Bbin sigue la estela de la función log(N) como hemos visto en teoría. No se aportan las funciones log(x) y x/2 en la gráfica debido a que sería coincidentes con Bbin y Blin respectivamente, lo que restaría claridad.

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



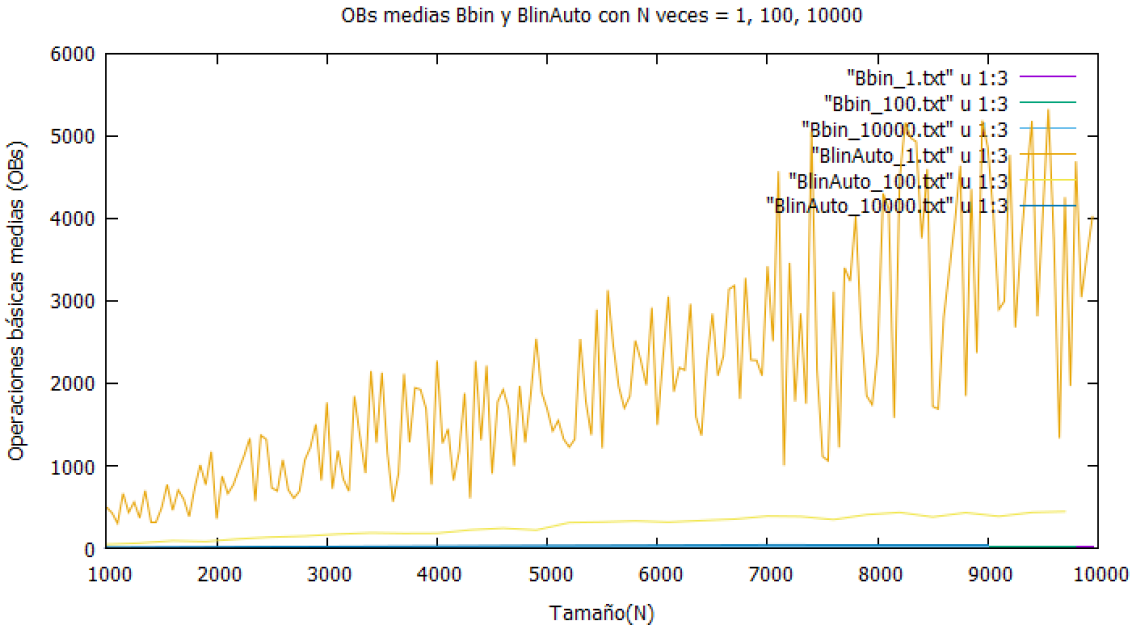
Comentarios:

Se muestra el tiempo promedio de reloj de búsqueda lineal (línea superior) y de búsqueda binaria (línea inferior).

Como cabía esperar después de analizar la gráfica de OB's medias el tiempo necesitado por Blin es mucho más grande que el utilizado por Blin. Podríamos decir que el tiempo de Blin es N/9 us (O(N)) y el de Bbin es del orden de log(N). No se ha representado f(x)=x/9 y g(x)=log(x) para envitar confusiones.

Comentar que el tiempo está representado en la gráfica en microsegundos para evitar trabajar con exponciales negativos, pero en la estructura PTIEMPO ha sido guardado en segundos tal y como se pedía en la documentación.

Gráfica comparando el número promedio de OBs entre la búsqueda binaria y la búsqueda lineal auto organizada (para los valores de n_veces=1, 100 y 10000), comentarios a la gráfica.

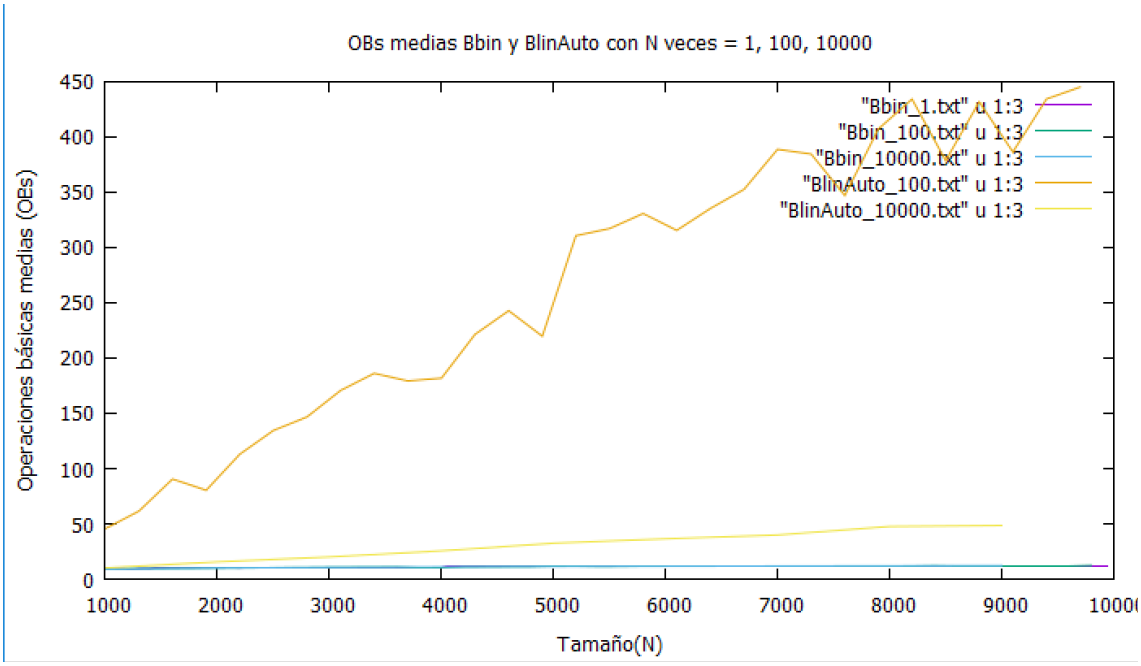


Comentarios:

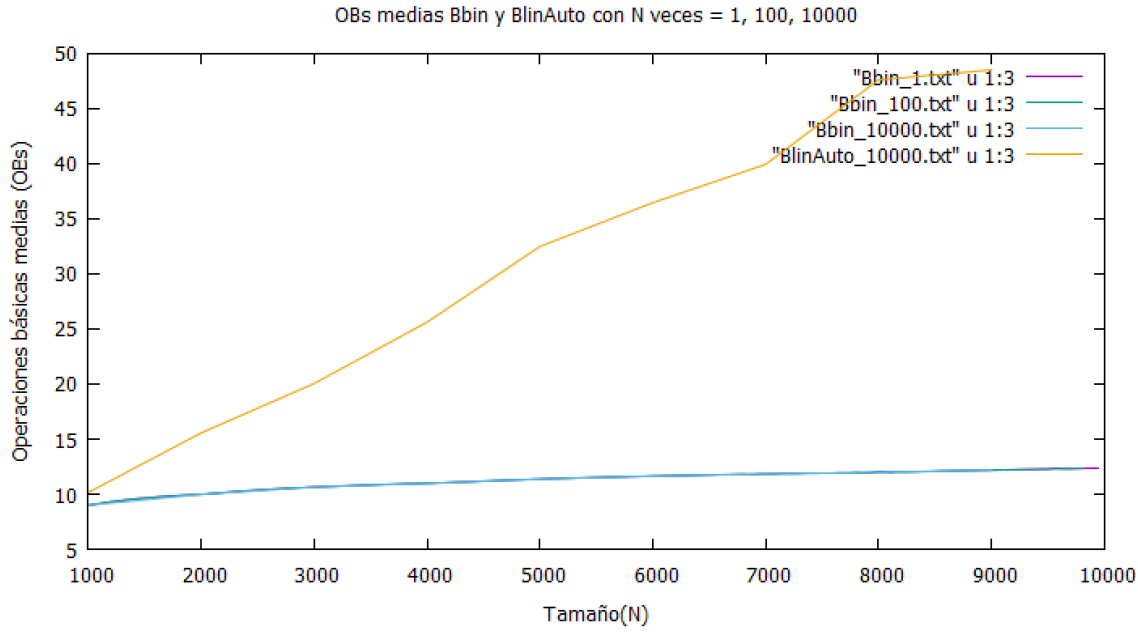
Se muestra el número promedio de OB´s de búsqueda lineal autoorganizada(línea superior y algunas inferiores) y de búsqueda binaria (varias líneas inferiores). Se aconseja seguir la leyenda de colores.

La primera impresión es que algo ha ido mal, pero después de reflexionar un poco se puede sostener que ese es el resultado que se debe obtener. La línea naraja supeior de tendecia muy variable se corresponde con Blin_auto con N_veces=1 y utilizando la función generadora de claves potencial. Como n_veces=1 la función Blin autoorganizada no ha tenido la oportunidad de recolocar las claves más buscadas al principio, lo que, junto con que se está utilizando la función generadora de claves potencial la cual es pseudoaleatoria (priorizando los números más bajos, pero lo que no tiene relevancia para n_veces=1) provoca una gran disparidad en las OB's dependiendo de si la clave está por el final, por el medio o por el principio.

Para mayor claridad, mostraremos la misma gráfica pero elimando la traza de BlinAuto con n_veces = 1.



Ahora nos encontramos aparentemente con el mismo problema que antes. Esto nos dice que utilizar la función de búsqueda lineal autoorganizada después de 100 búsquedas sigue siendo ineficiente comparada con otras opciones. Vamos a continuar elimando la traza de BlinAuto para n_veces = 100.



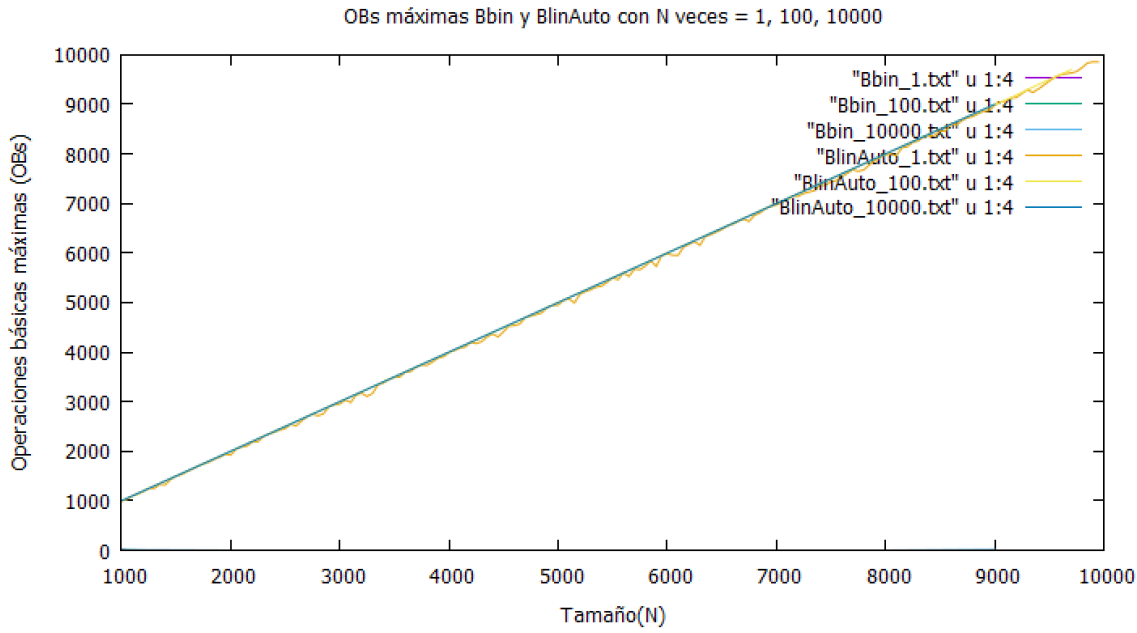
Con esta vista ya podemos percibir las gráficas de Bbin, las cuales están las tres superpuestas una encima de otra ya que la función Bbin tiene el mismo rendimiento independientemente del número de veces que hayamos buscado claves anteriormente.

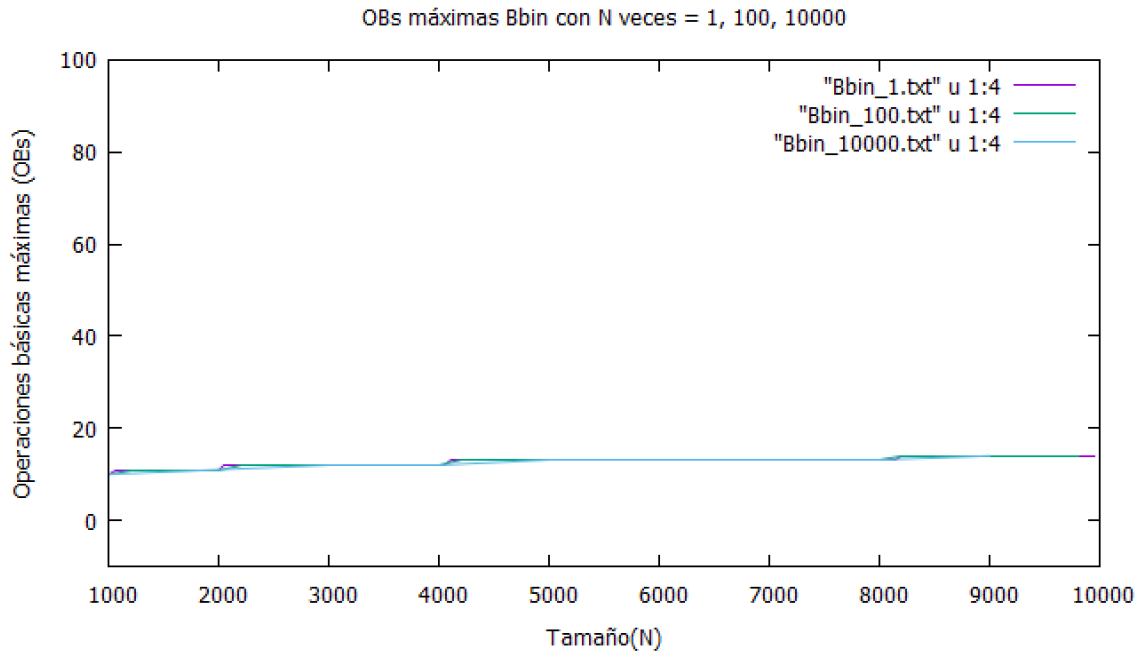
¿Pero con qué tenemos que quedarnos en este apartado? Sin lugar a dudas con la bestial mejora de rendimiento de BlinAuto según se van haciendo búsquedas previas. La escala del eje Y se ha reducido desde 5500 OB's a solo 50 en esta última. Simplemente brutal. Como conclusión podríamos decir que si necesitamos resultados inmediatos es mejor utilizar (a priori, porque aún tenemos gráficas que analizar) Bbin, pero si llegásemos a hacer una gran cantidad de búsquedas previas y con una frecuencia

potencial en la búsqueda de claves, es más eficiente BlinAuto ya que en el límite, tendería a encontrar las claves en O(1), lo cual es mejor que O(log(N)) de Bbin.

Comentar que esta disparidad de las trazas también se da en las futuras gráficas, por lo que se incluirán todas las gráficas necesarias al principio y se comentarán a posteriori. Se ruega prestar antención a la leyenda para saber lo que se está representando.

Gráfica comparando el número máximo de OBs entre la búsqueda binaria y la búsqueda lineal auto organizada (para los valores de n_veces=1, 100 y 10000), comentarios a la gráfica.





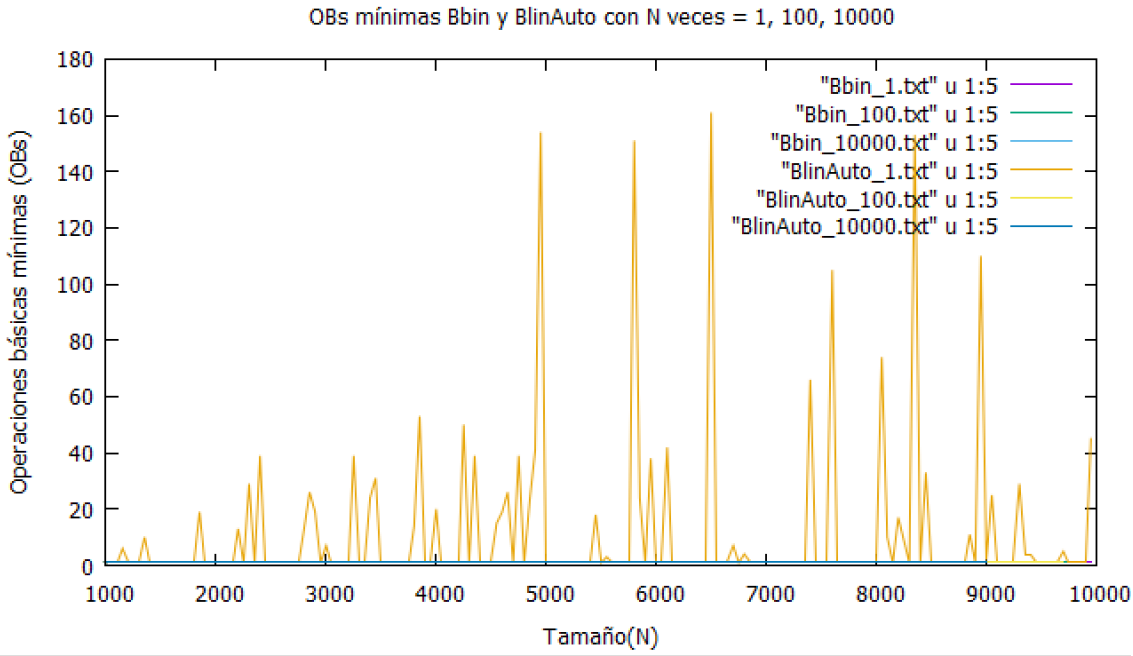
Comentarios:

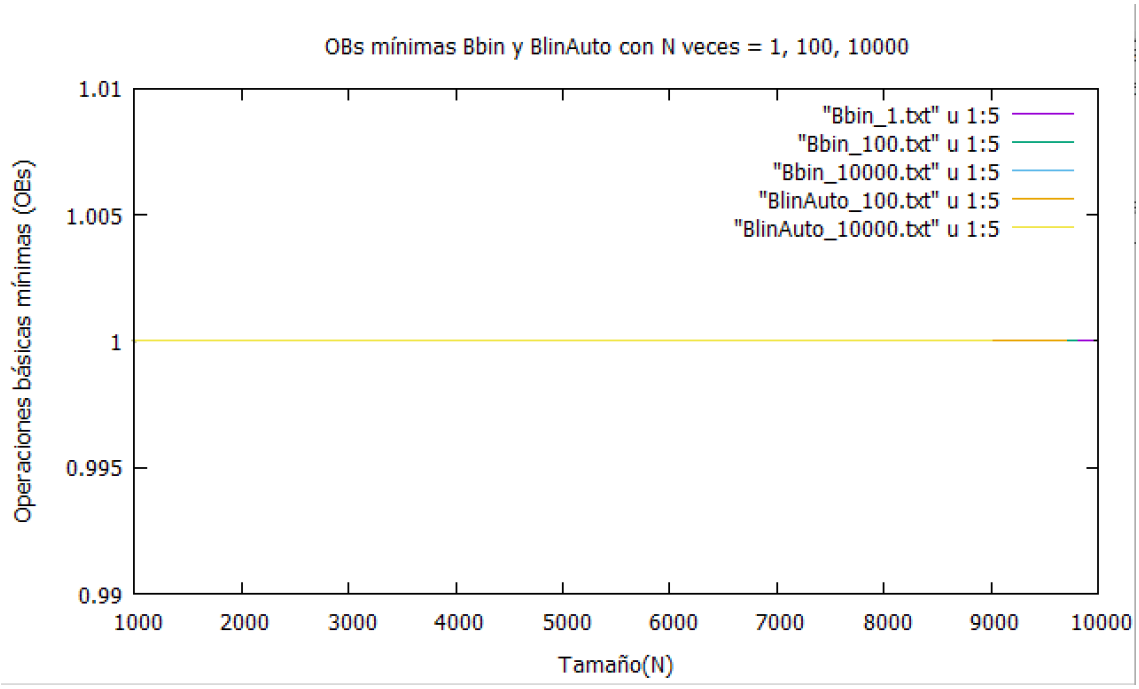
En la primera gráfica se muestran las OB's máximas de Bbin y BlinAuto para n_veces = 1, 100 y 10000. En la segunda se muestran únicamente las OB's máximas de Bbin para n_veces = 1, 100, 10000.

Se ha realizado esta descomposición porque las tres gráficas de BlinAuto se encuentran prácticamente superpuestas, debido a que antes de que BlinAuto tenga un redimiento óptimo, esta tiene que pasar por la fase en la que coloca los más buscados al principio, por lo que las OB's máximas van a coincidir independientemente del valor de n_veces.

Para poder observar más claramente la tendencia de Bbien se han eliminado estas tres trazas. Del mismo modo, Bbin también tiene superpuestas sus tres trazas, lo cual era de esperar porque en el propio caso del promedio de OB's en el apartado anterior ya se encontraban ceñidas una a la otra.

Gráfica comparando el número mínimo de OBs entre la búsqueda binaria y la búsqueda lineal auto organizada (para los valores de n_veces=1, 100 y 10000), comentarios a la gráfica.





Comentarios:

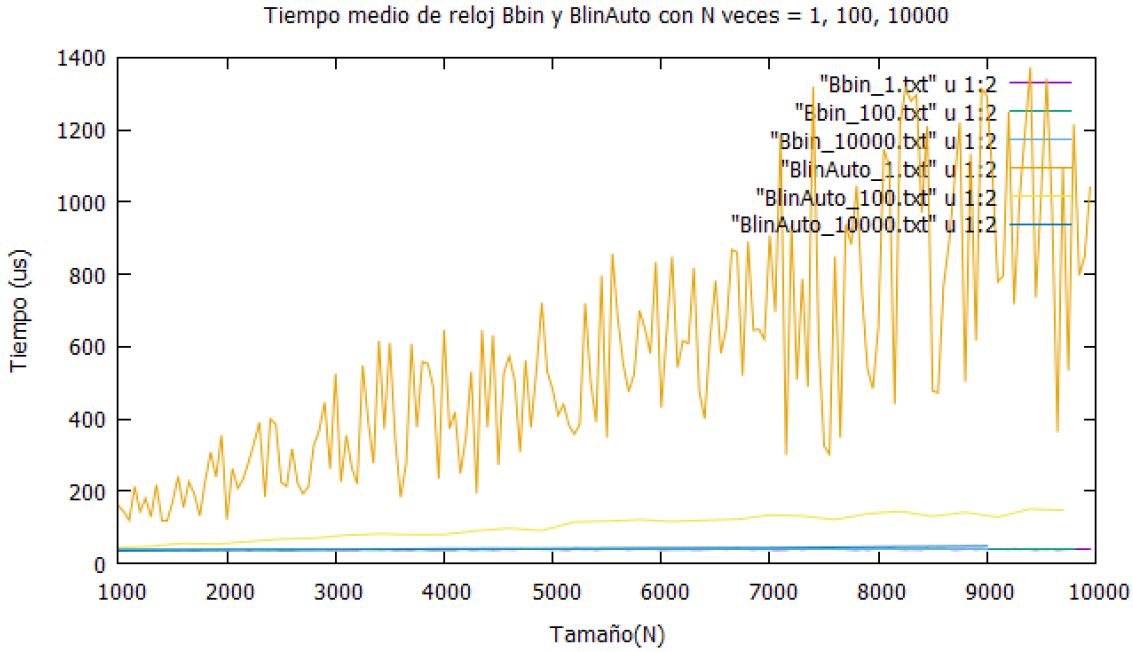
En la primera gráfica se muestran las OB's mínimas para Bbin y BlinAuto con n_veces = 1, 100 y 10000. En la segunda se ha suprimido la traza de BlinAuto con n_veces = 1 debido a que impedía percibir el resto.

No era muy difícil predicir que iba a pasar en este apartado. Cuando utilizamos BlinAuto con n_veces = 1 y con el generador de claves potencial tenemos una gran disparidad de número de OB's mínimas debido a la pseudoaleatoriedad del generador. BlinAuto no ha tenido suficientes búsquedas para recolocar los elementos más frecuentes al principio y junto con que las claves con mayor probabilidad no se encotraban en lugares bajos de la tabla en todos los casos(como es de esperar) las OB's mínimas se disparan.

Por otro lado, una vez que se ha ejecutado unas cuantas veces ya tiene bastantes opciones a encontrar la clave en la primera posición buscada. Debido a eso, BlinAuto con n_veces =100 y 10000 tiene un número de OB's mínimas igual a 1.

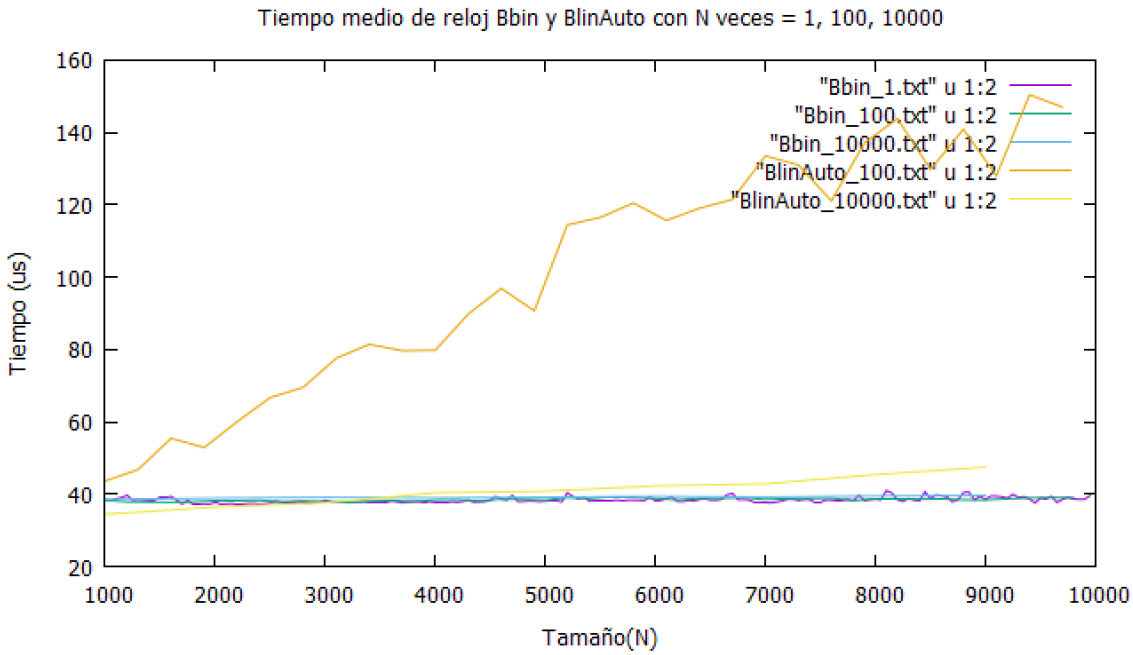
Por último, ya era de esperar que para todas las trazas de Bbin el nº de OB's mínimo iba a ser constatemente 1, ya que Bbin es eficiente buscando ya desde el primer momento y además, con Bbin se utiliza el generador de claves uniforme por lo que en alguna búsqueda se va a buscar por el elemento en la primera posición.

Gráfica comparando el tiempo medio de reloj entre la búsqueda binaria y la búsqueda lineal auto organizada (para los valores de n_veces=1, 100 y 10000), comentarios a la gráfica.

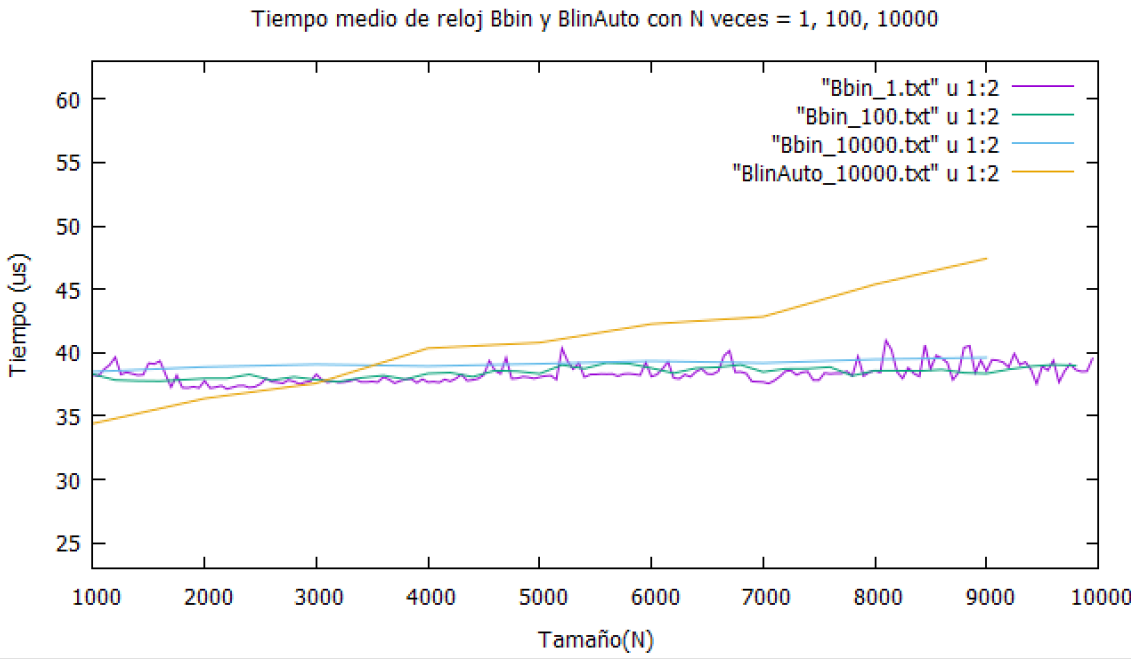


Comentarios:

Como ya viene siendo habitual, BlinAuto con n_veces = 1 produce una traza muy desigual, que demuestra lo ineficiente que es la función para pocas búsquedas. Vamos a eliminarla para percibir mejor el resto.



Ahora la gráfica ya tiene un aspecto más uniforme, pero aún así la traza de BlinAuto con n_veces = 100 es muy diferente al resto de trazas lo que vuelve a enseñarnos la ineficacia de esta función para pocas búsquedas comparada con otras como es Bbin. Por último, vamos a eliminar esta traza a ver que podemos contar.



Por último, esta gráfica nos aporta gran información. Podemos ver que para un tamaño de tabla pequeño (menor que 3000 elementos) y con una gran cantidad de búsquedas para permitir a BlinAuto ordenarse dependiendo de la frecuencia de las claves buscadas, su tiempo de ejecución es menor que cualquiera de las trazas de búsqueda binaria. No obstante según aumenta el tamaño, los casos en que la clave pseudoaleatoria del generador de claves potencial es una poco frecuente produce un tiempo de ejecución demasiado alto (tiene que buscar esa clave poco frecuente, por lo que estará cerca del final, en una tabla grande) para ser compensado por el tiempo cerca a O(1) de las claves frencuentes.

Sin embargo, de este apartado tenemos que sacar dos cosas en claro: primero, que blinAuto mejora considerablemente según se aumenta el número de búsquedas, ya que ha pasado de una escala de 1400 us a una de apenas 50 us; y segundo, que si aumentamos el número de búsquedas previas, podremos aumentar esa cota superior para la cual BlinAuto es más rápido que Bbin. En este caso la cota fue de N=3000, pero si aumentamos n_veces, no sería de extrañar que la cota fuese lo suficientemente óptima como para no tener que usar más Bbin (muy difícil pero posible). Esa el la ventaja de BlinAuto (para generador de claves potencial), que puede llegar a la O(1) de eficiencia, comparada con Bbin que tiene una eficiencia estable de O(log(N)).

5. Respuesta a las preguntas teóricas.

5.1 Pregunta 1

En todos los algoritmos nos encontramos con un bucle while o for, por lo que la OB tendrá que estar dentro del bucle y ser representativa respecto a lo que hace en el algoritmo. Dentro de los bucles de las tres funciones siempre hay un elemento que se repite: un if que tiene como condición la comparación de clave. En el caso de la búsqueda lineal es necesario hacer dos comparaciones de clave dentro de cada bucle, pero siempre que nos encontramos con un if nos encontramos con una comparación de clave. Lo que hay dentro de un if no puede ser candidato a ser una OB, ya que no siempre se entrará dentro del if, sin embargo siempre se llevará a cabo la comparación del if.

Por ello, nuestra OB elegida (al igual que en teoría) es la comparación de clave.

5.2 Pregunta 2

En la búsqueda lineal y en la búsqueda binaria, el mejor caso se da cuando se encuentra la clave en la primera cdc. Con esto concluimos que:

$B_{bb}(n)=B_{bl}(n)=1=O(1)$

En búsqueda lineal, el peor caso se da cuando el término a encontrar es el último de la tabla, es decir, aquel que encontramos después de haber hecho una cdc con cada elemento de la tabla. Esto equivale a escribir:

$W_{bl}(n)=n=O(n)$

En búsqueda binaria, podemos ver que la búsqueda es equivalente a buscar un número en un heap, por lo que el máximo número de comparaciones de clave que hay que hacer corresponde con la profundidad de este heap. Teniendo en cuenta que la profundidad de un heap es floor(log(n)):

$W_{bb}(n)=floor(log(n))=O(log(n))$

5.3 Pregunta 3

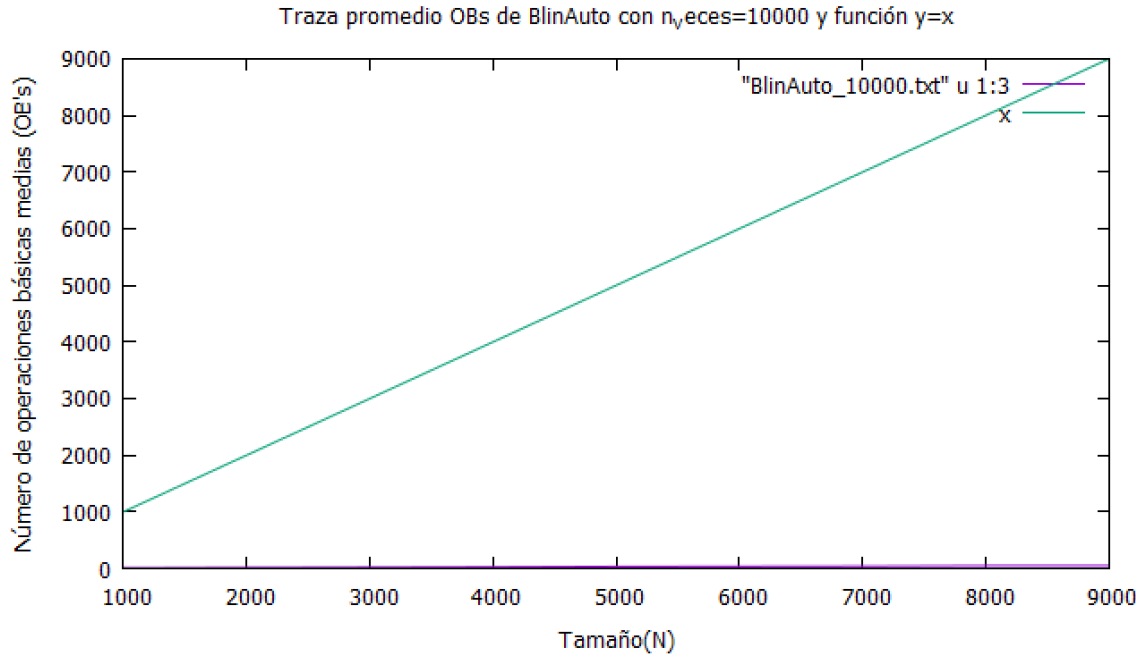
Los elementos más buscados realizan más veces un *swap* con su anterior, por lo que lo normal es que estén en las primeras posiciones de la tabla, es decir, en las primeras posiciones que se visitan cuando se busca una clave. Por el contrario, aquellos elementos de la tabla que nunca o casi nunca sean buscados quedarán relegados al final de la tabla, lugar donde tiene más coste buscar, pero que se realizará pocas veces, ya que allí quedan relegadas las claves que aparecen con poca frecuencia.

Por si queda alguna duda se pueden revisar las gráficas del capítulo anterior y sus comentarios. Todo esto nos da una muy buena idea de como funciona BlinAuto.

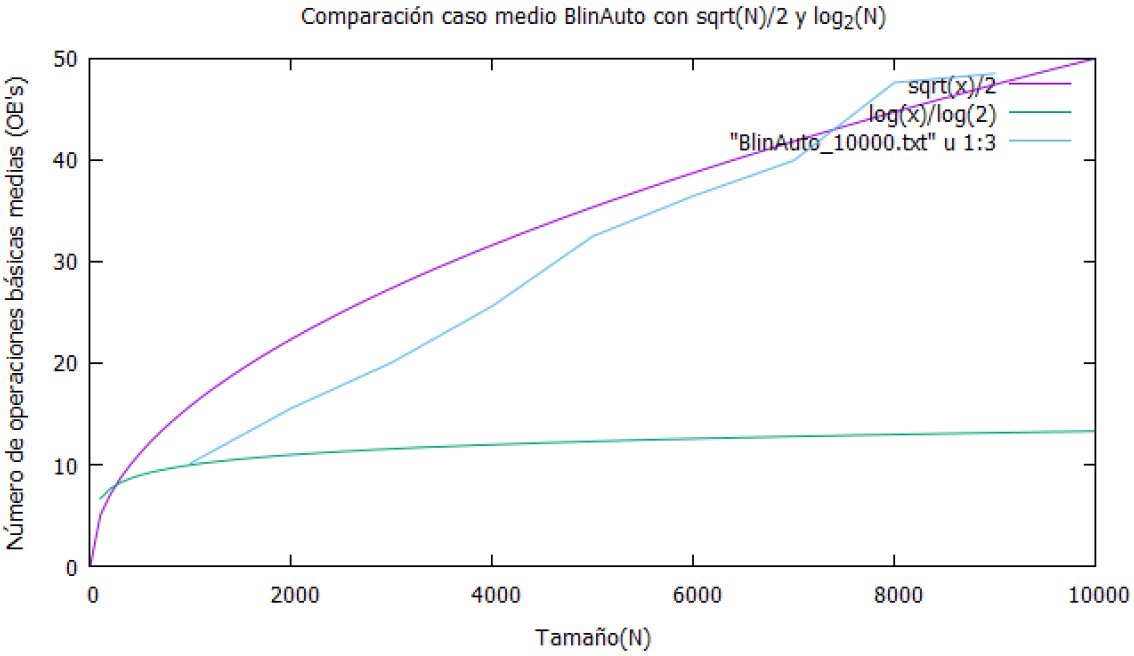
5.4 Pregunta 4

Lo primero, debemos destacar que el orden de ejecución medio depende siempre del algoritmo dado, en esta ocasión utilizaremos generador_claves_potencial. Como también se nos ha indicado, no debemos deducir teóricamente un algoritmo, sino que vale con ver que se aproxima correctamente a una función que nos valdrá como caso medio.

Por lo tanto, vamos a probar empíricamente cual es la función que mejor se ajusta al promedio de OB's obtenidas en BlinAuto para n_veces=10000, para el cual ya se han realizado un número considerable de búsquedas.



Como se puede ver en la gráfica, el promedio de OB's de BlinAuto no es O(N). Vamos a probar con funciones más pequeñas.



En esta gráfica hemos incluído la raíz cuadrada de X partido de 2 y el logaritmo binario de X. A priori, la función a promediar parece semejarse más a sqrt(x)/2 lo que sería O(sqrt(N)). No obstante, tenemos que tener en cuenta que la traza para BlinAuto que estamos utilizando es después de 10000*10000 búsquedas, es un número grande, pero no imposible superar, por lo que podemos esperar que esa traza fueses más baja en otros casos.

En conclusión, podríamos decir que el orden de ejecución medio de blin_auto es del orden de O(log(N)) o O(sqrt(N)) y claramente inferior a O(N). Ambas estimaciones son aceptables, teniendo tanto en cuenta lo teórico como lo empírico.

5.5 Pregunta 5

Vamos a demostrar que funciona la búsqueda binaria por inducción fuerte. Demostraremos dos casos base (el primero es trivial): n=1,2. Cabe destacar que n es el número de elementos que componen la lista. Da igual cuales sean mientras estén ordenados, en este caso tomaremos sin pérdida de generalidad el caso particular en el que P=1 y los número son escogidos con un intervalo de 1 hasta llegar a U, inclusive.

Cuando n=1, P=1 y U=1, se compara la clave con el contenido de la tabla en M=1, si es la correcta se devuelve que la clave ha sido encontrada en la posición 1, si no ha sido encontrada se continúa con el algoritmo. Si la clave es menor que el contenido de la tabla en M, P=1, U=0, se llega a la condición de parada, la clave no ha sido encontrada. Si la clave es mayor que el contenido de la tabla en M, P=2 y U=1, se llega a la concición de parada, la clave no ha sido encontrada. El algoritmo es correcto para n=1.

Cuando n=2, P=1 y U=2, se compara la clave con el contenido de la tabla en M=1, si es la correcta se devuelve que la clave ha sido encontrada en la posición 1, si no ha sido encontrada se continúa con el algoritmo. Si la clave es menor que el contenido de la tabla en M, P=1, U=0, se llega a la condición de parada, la clave no ha sido encontrada. Si la clave es mayor que el contenido de la tabla en M, P=2 y U=2, podemos volver al caso 1, pues es análogo. El algoritmo es correcto para n=2.

Hipótesis de inducción: el algoritmo de búsqueda binaria es correcto para n=1,2, …,r-1

Demostramos el algoritmo para n=r, donde P=1 y U=r. M es, por lo tanto, floor((1+r)/2). Si el contenido de la tabla en M es la clave, se ha encontrado la posición de la clave en la tabla. Si la clave es menor que el contenido de la tabla en M, P=1, U=M-1, se sigue aplicando el algoritmo. Como U-P < r-1 (cantidad de elementos), este caso ya está demostrado por hipótesis de inducción. Si la clave es mayor que el contenido de la tabla en la posición M, P=M+1 y U=r. Como U-P < r-1, este caso ya queda demostrado por hipótesis de inducción. El algoritmo es correcto para n=r, el algoritmo queda demostrado por inducción para todos los enteros positivos.

6. Conclusiones finales.

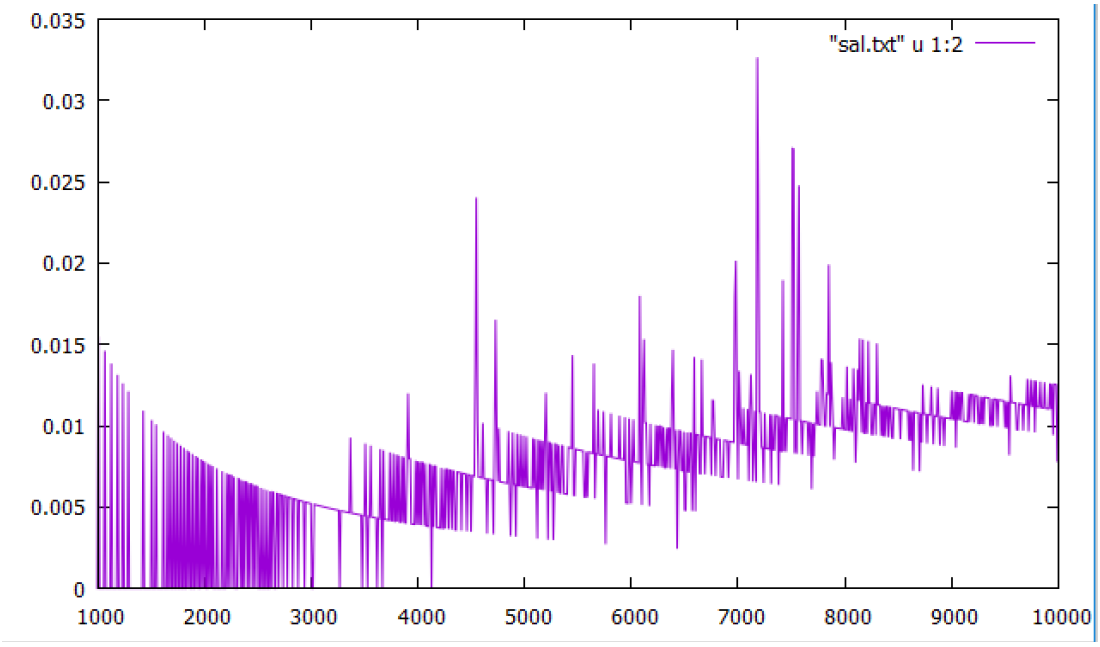
Como última práctica del curso de Análisis de Algoritmos no se esperaba menos. La práctica junta todo lo visto durante el curso, tanto como estimaciones de funciones, algoritmos de ordenación, producción de OB's y tiempos de forma empírica y por último, algoritmos de búsqueda y su análisis.

En la primera parte se han programado las funciones de diccionario y, más importante, las funciones de búsqueda (búsqueda lineal, búsqueda binaria y búsqueda lineal autoorganizada).

En la segunda parte de la práctica, se han implementado las funciones de medición de tiempos, que teníamos ya hechas para los algoritmos de ordenación, pero ahora las hemos adaptado a los algoritmos de búsqueda. Las gráficas de tiempos y OB's nos han aportado información importante de los algoritmos propuestos.

Esta práctica nos ha permitido acercarnos al uso de los algoritmo de búsqueda de claves, desde algoritmo “normales” como es búsqueda lineal, el cual no es muy eficiente para tablas de gran tamaño. Debido a esto no se ha intentado producir tiempos y/o OB's para Blin con n_veces = 10000. Por otro lado, nos a permitido comparar las limitaciones de búsqueda lineal con otros algoritmos más eficaces como búsqueda binaria. No podemos resaltar tanto la función búsqueda lineal autoorganizada debido a que se tiene que cumplir una premisa muy importante: búsqueda de claves NO uniforme y con una clara tendencia hacia unas ciertas claves. Sin esta premisa no es de gran utilidad esta función ya que se asemejaría bastante a búsqueda lineal.

Técnicamente no es una práctica difícil pero sí que se han tenido que superar ciertos problemas y complejidades durante la práctica. Por ejemplo, misteriosamente la función de clock() ha dejado de funcionar correctamente con respecto a otras prácticas, produciendo resultados en las mediciones de tiempos anormales, por lo que hemos tenido que cambiar nuestra implementación de tiempos_medio_busqueda utilizando la otra función de medición de tiempo, clock_gettime. A modo de curiosidad, esta es la salida de búsqueda lineal con un diccionario de tamaño 1000 – 10000 en incremento de 50 en 50, con n_veces=1 con la función clock() que estaba dando problemas:



Parece que sigue un patrón determinado, el cual nos ha traído varios dolores de cabeza y el cual no hemos conseguido dar explicación. Parece como si hubiese una cota superior para el tiempo de ejecución para tamaños pequeños y que se transforma en una cota inferior para el tiempo de tablas de mayor tamaño, dejando lugar a una nueva cota superior (y continuando el ciclo).

Como conclusión final, ha sido una buena práctica para terminar el curso, pero aún así se hecha en falta otros algoritmos de ordenación como HeapSort u otros de búsqueda como la utilización de árboles de búsqueda etc...

Estaría bien que para futuras prácticas se centrasen un poco más en la implementación de estos algoritmos y no tanto en el análisis de unos pocos.