Análisis de Algoritmos 2020/2021

Práctica 3

Eduardo Terrés y Víctor Pérez

Grupo 06

Código	Gráficas	Memoria	Total

1. Introducción

La finalidad de la práctica es llevar a cabo un análisis sobre distintos algoritmos de búsqueda, en tablas ordenadas y no ordenadas. De esta forma, será necesario crear un TAD diccionario, donde se puedan almacenar los datos susceptibles de búsqueda, e incluya la funcionalidad de decidir su posición relativa - es decir, si están o no ordenados -. Una vez implementadas las funciones para cada método de búsqueda, será cuestión de tomar los datos con las funciones de toma y recogida de información, del fichero *tiempos.c*. Este último paso incluye la posterior generación de gráficas y su análisis.

2. Objetivos

A continuación, se explicará de forma resumida la finalidad que existe detrás de las implementaciones realizadas. Como es evidente, la información será complementada con el resto de puntos de la memoria. Así pues, el propósito de esta sección es reflejar el problema tal y como se nos planteó.

2.1 Apartado 1

En primer lugar, y como ya se mencionó en la introducción, la creación del TAD diccionario permite la manipulación de los datos a la hora de llevar a cabo los procedimientos de búsqueda. Las funciones con las que cuenta son las siguientes:

```
PDICC ini_diccionario (int tamanio,char orden);

void libera_diccionario(PDICC pdicc);
```

Las funciones de creación y liberación son triviales, aunque imprescindibles, respecto de la funcionalidad que aportan al TAD.

```
int inserta_diccionario(PDICC pdicc,int clave);

int insercion_masiva_diccionario (PDICC pdicc,int *claves,int n_claves);
```

Los prototipos anteriores corresponden a las funciones que tienen como finalidad la inserción de claves en el diccionario introducido por parámetros de entrada. La segunda facilita esta tarea para una cantidad de datos masiva.

Finalmente, y en lo que respecta al TAD, la siguiente función permite buscar una clave en un diccionario creado, haciendo uso del *método* introducido por parámetros de entrada.

```
int busca_diccionario(PDICC pdicc,int clave,int *ppos,pfunc_busqueda metodo);
```

Asimismo, se pide implementar tres funciones de búsqueda de claves en un diccionario. En primer lugar, el algoritmo de búsqueda binaria para diccionarios ordenados, con el siguiente prototipo:

```
int bbin(int *tabla,int P,int U,int clave,int *ppos);
```

El otro algoritmo a implementar es el de búsqueda lineal. Además, se pide la implementación de una variante, búsqueda lineal autoorganizada, que modifica la tabla para reducir el coste de búsqueda de claves visitadas con mayor frecuencia.

```
int blin(int *tabla,int P,int U,int clave,int *ppos);

int blin_auto(int *tabla,int P,int U,int clave,int *ppos);
```

El retorno de todos los métodos de búsqueda es el número de OBs realizadas, para poder efectuar el estudio; ERR en caso de error.

La función del TAD que gestiona la búsqueda de elementos a través del método introducido por parámetros de entrada es la siguiente:

```
int busca_diccionario(PDICC pdicc, int clave, int *ppos, pfunc_busqueda metodo)
```

2.2 Apartado 2

El *apartado 2* parte de las rutinas del *apartado 1*, y tiene como finalidad analizar su eficiencia, en términos de costes temporales. Es así que se pide crear un conjunto de funcionalidades que permitan, dada un método, iterarlo sobre un diccionario para extraer resultados que puedan ser comparados y representados a posteriori. De esta forma, los prototipos de las funciones a implementar son:

```
short tiempo_medio_busqueda(pfunc_busqueda metodo,
        pfunc_generador_claves generador,
        char orden, int N, int n_veces, PTIEMPO ptiempo);
```

La función *tiempo_medio_busqueda* estudia el método para el tamaño de tabla introducido por parámetros de entrada - *N* -, y busca cada clave de la lista generada mediante *generador* *n_veces* veces.

La función principal y que es llamada en primer lugar es la siguiente:

```
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)
```

Su finalidad es hacer llamadas a *tiempo_medio_busqueda* para los distintos tamaños de diccionario. Finalmente, esta última función almacena los datos generados en un fichero de texto.

```
short guarda_tabla_tiempos(char* fichero, PTIEMPO tiempo, int n_tiempos)
```

La conexión que existe entre ellas se explicará en el *apartado 3.2*.

3. Herramientas y metodología

La práctica ha sido realizada en un entorno de ejecución de *Linux*, utilizando *gcc* como compilador del lenguaje *C*, *valgrind* como revisor de la gestión de la memoria, y *Visual Studio Code* y *Netbeans* como IDEs. El control de versiones se ha realizado mediante *git* y a través de un repositorio privado en *Github*. Las gráficas se han realizado mediante *scripts* del programa *Gnuplot*.

3.1 Apartado 1

3.1.1 TAD Diccionario

La programación del tipo abstracto de datos es directa. De esta forma, las funciones de creación y liberación de diccionarios, así como la búsqueda no plantean grandes complicaciones a nivel de lógica interna. El control de errores de *ini_diccionario* es el único que hace uso de *if* en vez de *assert*, ya que los parámetros introducidos pueden ser erróneos y no han sido consultados previamente. En contraposición, el control de errores – que respecta al parámetro *pdicc* - de las funciones que dependen de un diccionario se ha implementado mediante *asserts*, ya que se entiende que los parámetros deben estar inicializados y en el rango de valores correcto.

En el caso de la función *inserta_diccionario*, cabe destacar que la algoritmia que subyace a la inserción de claves en un diccionario ordenado es análoga al método de ordenación *InsertSort*, aunque adaptada para posicionar correctamente un único elemento. El control de errores de esta última, así como *inserción_masiva_diccionario* de las variables de tipo *array* se ha efectuado mediante *if*, para que el programa que hace la llamada en primer lugar pueda interpretar el retorno *ERR*. El resto de comprobaciones se efectúa haciendo uso de *asserts*, de tal forma que se aporta un nivel alto de abstracción para el TAD.

Para poder comprobar el funcionamiento del tipo de datos diccionario hemos creado un programa *ejercicio0.c*, que tiene como finalidad crear e insertar una permutación en un diccionario.

3.1.2 Funciones de búsqueda

Una vez conocido el funcionamiento del algoritmo, hemos implementado la función que busca una clave en un diccionario haciendo uso de la búsqueda lineal. Para el correcto funcionamiento del método se asume como premisa que la tabla esta ordenada por sus claves. De esta forma, el método consiste en comparar el valor buscado con la mitad de la tabla, dividirla en dos partes y repetir el proceso para la subtabla donde se debería encontrar el elemento buscado en función del resultado de la comprobación inicial de la iteración. A nivel de implementación, las subtablas se obtienen modificando el subíndice y superíndice del vector de variables *tabla* y repitiendo el procedimiento para los nuevos valores que quedan tras cada iteración.

Análogamente, el algoritmo de búsqueda lineal se ha implementado siguiendo la lógica trivial del mismo. El funcionamiento es recorrer la tabla de forma secuencial e ir comparando el elemento de la posición en cuestión con el valor buscado, hasta que se encuentre o se termine la tabla. Una diferencia respecto de la búsqueda binaria es que la tabla no debe estar necesariamente ordenada.

En último lugar, la búsqueda lineal autoorganizada sigue la misma lógica que la búsqueda lineal, pero añadiendo la modificación de la tabla. Cabe destacar que, si se ha realizado el swap, la posición de la clave encontrada – almacenada en **ppos* - será la siguiente a la posición en la que se encuentra la clave tras la función. En otras palabras, se devuelve el elemento donde se encontró en un primer lugar. Para el *swap* no se ha implementado una función auxiliar, porque el intercambio de elementos se puede resolver con dos asignaciones y se emplea una única vez en la rutina.

A modo de comentario general, la posición devuelta en el parámetro **ppos* pertenece al intervalo [0,U], de forma que para una lista de números ordenada, el número *i* se encuentra en la posición *i-1* de la tabla.

Para poder comprobar el funcionamiento de las funciones de búsqueda, hemos hecho uso del programa *ejercicio1.c*, que permite comprobar la búsqueda de un elemento en un diccionario ordenado o desordenado, y el número de OBs llevadas a cabo.

3.2 Apartado 2

La función principal y que recibe la llamada en primer lugar es *genera_tiempos_busqueda*. Como ya se ha mencionado, su función es hacer llamadas a *tiempo_medio_busqueda* para los distintos tamaños de diccionario en el intervalo e incremento indicados por parámetros de entrada, y una última llamada para almacenar los datos. Cabe destacar que, en caso de que el incremento no genere intervalos de igual longitud, la función hace un ajuste mediante la siguiente linea de código, de forma que los resultados a obtener tienen una distancia uniforme respecto a su tamaño:

```
num_max = num_max - (num_max - num_min) % incr;
```

La función *tiempo_medio_busqueda* alberga gran parte la lógica que subyace al análisis de un método: la iteración del mismo sobre el diccionario, y la recolección de datos específicos y de valor para el estudio. El puntero de estructura tipo TIEMPO **ptiempo* se emplea para almacenar los datos. Finalmente, una vez se tienen todos los datos resultantes de las consecutivas llamadas para los distintos tamaños, son almacenados en un fichero por la función *guarda_tabla-_tiempos.*

El control de errores de la función principal - *genera_tiempos_busqueda* – incluye *asserts* para los parámetros asociados a índices y variables asociadas al diccionario, e *if* para las funciones empleadas y el fichero, siguiendo la lógica previamente mencionada de permitir a la función que llama al método interpretar el error y decidir sobre el trascurso de la ejecución. En lo que respecta a las otras dos funciones, se entiende que todos los parámetros han sido previamente comprobados, de ahí el uso de *asserts.*

Como las funciones efectúan un número alto de comprobaciones tras las cuales, en caso de error, se detiene la ejecución del programa, hemos implementado una rutina auxiliar *libera_memoria* que sirve para liberar la memoria asociada a un diccionario y dos vectores de variables de enteros que almacenan la permutación de la tabla diccionario y las claves.

Para poder comprobar el funcionamiento de estas rutinas, hemos hecho uso del programa *ejercicio2.c*, así como de las gráficas obtenidas en la sección 5.

4. Código fuente

4.1 Apartado 1

```c
/**
 * Autores: Eduardo Terres y Victor Perez
 * Fecha: 20/11/2020
 *
 * Descripcion: La funcion crea un diccionario de tipo orden
 * y con un tamanio maximo de elementos introducido por
 * parametro de entrada
 *
 * Parametros entrada:
 *   -int tamanio: numero de elementos maximos que admite el
 *    diccionario
 *   -int orden: tipo de diccionario. toma valores ORDENADO o
 *    NO_ORDENADO
 * Retorno:
 * La funcion retorna el diccionario creado; NULL en caso de
 * error
 **/
PDICC ini_diccionario(int tamanio, char orden) {
  PDICC dic = NULL;
  int *tabla = NULL;

  /* CdE */
  /* No admite un diccionario vacio */
  if (tamanio < 1)
    return NULL;
  if (orden != ORDENADO && orden != NO_ORDENADO)
    return NULL;

  dic = (PDICC)malloc(sizeof(dic[0]));
  if (dic == NULL)
    return NULL;

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

  /* Tabla */
  dic->tabla = (int *)malloc(tamanio * sizeof(tabla[0]));
  if (dic->tabla == NULL) {
    free(dic);
    return NULL;
  }
  return dic;
}
```

```c
/**
 * Autores: Eduardo Terres y Victor Perez
 * Fecha: 20/11/2020
 *
 * Descripcion: La funcion libera toda la memoria a
sociada
 * a un diccionario.
 *
 * Parametros entrada:
 *   -
PDICC pdicc: diccionario que se quiere liberar
 * Retorno:
 * void
 **/
void libera_diccionario(PDICC pdicc)
{
  if (pdicc != NULL)
  {
    if (pdicc->tabla != NULL)
      free(pdicc->tabla);
    free(pdicc);
  }
  return;
}
```

```c
/**
 * Autores: Eduardo Terres y Victor Perez
 * Fecha: 20/11/2020
 *
 * Descripcion: La funcion inserta una clave en un diccionar
io,
 * respetando la ordenacion del mismo
 *
 * Parametros entrada:
 *   -PDICC pdicc: diccionario donde se inserta la clave
 *   -int clave: valor a introducir en el diccionario
 * Retorno:
 * ERR en caso de error; OK en caso contrario
 **/
int inserta_diccionario(PDICC pdicc, int clave)
{
  int i, a;

  /* CdE */
  if(pdicc == NULL) return ERR;
  assert(pdicc->orden == ORDENADO
        || pdicc->orden == NO_ORDENADO);
  /* Diccionario completo */
  if(pdicc->n_datos == pdicc->tamanio) return ERR;

  /* Inserta la clave en la ultima posicion */
  pdicc->tabla[pdicc->n_datos] = clave;

  /* Caso diccionario ordenado */
  if (pdicc->orden==ORDENADO) {
    a = pdicc->tabla[pdicc->n_datos];
    i = pdicc->n_datos - 1;
    while (i>=0 && a<pdicc->tabla[i]) {
      pdicc->tabla[i+1] = pdicc->tabla[i];
      i--;
    }
    pdicc->tabla[i+1] = a;
  }
  pdicc->n_datos++;

  return OK;
}
```

```c
/**
 * Autores: Eduardo Terres y Victor Perez
 * Fecha: 20/11/2020
 *
 * Descripcion: La funcion inserta una clave en un
diccionario,
 * respetando la ordenacion del mismo
 *
 * Parametros entrada:
 *   -
PDICC pdicc: diccionario donde se quiere insertar l
as claves
 *   -int *claves: array que contiene las claves
 *   -int n_claves: numero de claves a insertar
 * Retorno:
 * La funcion retorna el numero total de veces que
se ejecuta la OB
 * - cdc - de inserta_diccionario; ERR en caso de e
rror.
 **/
int insercion_masiva_diccionario(PDICC pdicc, int *
claves, int n_claves)
{
  int i, ret = 0, cont = 0;

  /* CdE */
  if (pdicc == NULL || claves == NULL)
    return ERR;
  assert(n_claves >= 1);
  assert(pdicc->orden == ORDENADO || pdicc->orden
== NO_ORDENADO);

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

  return cont;
}
```

```c
/**
 * Autores: Eduardo Terres y Victor Perez
 * Fecha: 20/11/2020
 *
 * Descripcion: La funcion busca una clave en un diccionario
 * introducido por pantalla. La funcion asume que los parametros
 * de entrada han sido comprobados previamente
 *
 * Parametros entrada:
 *   -PDICC pdicc: diccionario donde se inserta la clave
 *   -int clave: valor a introducir en el diccionario
 *   -
int *ppos: puntero a variable de tipo entero. su estado inicial
 *    es irrelevante, pero debe estar declarado.
 *   -
pfunc_busqueda metodo: algoritmo empleado para la busqueda
 * Retorno:
 *   -
int *ppos: puntero a variable de tipo entero, donde se almacena
 *    la posicion del elemento buscado.
 * Retorna el numero de operaciones básicas realizadas por metodo;
 * ERR en caso de error.
 **/
int busca_diccionario(PDICC pdicc, int clave, int *ppos, pfunc_busqueda metodo)
{
  /* Previamente comprobados */
  assert(pdicc != NULL);
  assert(ppos != NULL);
  assert(metodo != NULL);
  return metodo(pdicc->tabla, 0, pdicc->n_datos - 1, clave, ppos);
}
```

```c
/* Funciones de busqueda del TAD Diccionario */
/**
 * Autores: Eduardo Terres y Victor Perez
 * Fecha: 27/11/2020
 *
 * Descripcion: La funcion busca una clave en un diccionario
 * introducido por pantalla. Hace uso del algoritmo de busqueda
 * binaria. Las tabla de entrada debe estar ordenada.
 *
 * Parametros entrada:
 *   -
int *tabla: array de enteros donde se encuentran los elementos
 *    susceptibles de busqueda
 *   -
int *ppos: puntero a variable de tipo entero. su estado inicial
 *    es irrelevante, pero debe estar declarado.
 *   -int P: indice inferior a partir del cual se busca la clave
 *   -int U: indice superior hasta el que se busca la clave
 * Retorno:
 *   -
int *ppos: puntero a variable de tipo entero, donde se almacena
 *    la posicion del elemento buscado.
 *    IMPORTANTE: Los indices empiezan a contar desde el 0
 * Retorna el numero de operaciones básicas realizadas por metodo;
 * ERR en caso de error.
 **/
int bbin(int *tabla,int P,int U,int clave,int *ppos)
{
  int medio, cmp;
  int cont = 0;

  /* CdE */
  assert(tabla != NULL);
  assert(P >= 0);
  assert(U >= P);
  assert(ppos != NULL);

  while(P <= U){
    medio = (P + U) / 2;
    cmp = tabla[medio] - clave;

    cont ++;
    /* Clave encontrada */
    if(cmp == 0){
      *ppos = medio;
      return cont;
    }

    if(cmp > 0){
      /* Subtabla izquierda */
      U = medio - 1;
    }else{
      /* Subtabla derecha */
      P = medio + 1;
    }
  }

  /* Elemento no encontrado */
  *ppos = NO_ENCONTRADO;
  return cont;
}
```

```c
/**
 * Autores: Eduardo Terres y Victor Perez
 * Fecha: 27/11/2020
 *
 * Descripcion: La funcion busca una clave en un diccionario
 * introducido por pantalla. Hace uso del algoritmo de busqueda
 * lineal.
 *
 * Parametros entrada:
 *   -int *tabla: array de enteros donde se encuentran los elementos
 *    susceptibles de busqueda
 *   -int clave: valor a introducir en el diccionario
 *   -int *ppos: puntero a variable de tipo entero. su estado inicial
 *    es irrelevante, pero debe estar declarado.
 *   -int P: indice inferior a partir del cual se busca la clave
 *   -int U: indice superior hasta el que se busca la clave
 * Retorno:
 *   -int *ppos: puntero a variable de tipo entero, donde se almacena
 *    la posicion del elemento buscado; NO_ENCONTRADO si clave
 *    no se encuentra en la tabla.
 *    IMPORTANTE: Los indices empiezan a contar desde el 0
 * Retorna el numero de operaciones básicas realizadas por metodo;
 * ERR en caso de error.
 **/
int blin(int *tabla,int P,int U,int clave,int *ppos)
{
  int i, cont = 0;

  /* CdE */
  assert(tabla != NULL);
  assert(P >= 0);
  assert(U >= P);
  assert(ppos != NULL);

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

  /* Elemento no encontrado */
  *ppos = NO_ENCONTRADO;
  return cont;
}
```

```c
/**
 * Autores: Eduardo Terres y Victor Perez
 * Fecha: 27/11/2020
 *
 * Descripcion: La funcion busca una clave en la tabla pasada como argumento que este entre
 * los indices P y U. Emplea el algoritmo de búsqueda lineal autoorganizada:
 * si se encuentra la clave, se intercambia con el elemento en la posición anterior
 * siempre y cuando este no sea el primero. De esta manera las claves más buscadas se
 * encontraran primero.
 * IMPORTANTE: si la tabla esta ordenada, se desordenara tras la llamada a la funcion.
 *
 * Parametros entrada:
 *   -int *tabla: array de enteros donde se encuentran los elementos
 *    susceptibles de busqueda
 *   -int clave: valor a introducir en el diccionario
 *   -int *ppos: puntero a variable de tipo entero. su estado inicial
 *    es irrelevante, pero debe estar declarado.
 *   -int P: indice inferior a partir del cual se busca la clave
 *   -int U: indice superior hasta el que se busca la clave
 * Retorno:
 *   -int *ppos: puntero a variable de tipo entero, donde se almacena
 *    la posicion del elemento buscado; NO_ENCONTRADO si clave
 *    no se encuentra en la tabla.
 *    IMPORTANTE: si se ha realizado el swap, esta posición será la
 *    siguiente a la posición en la que se encuentra la clave tras
 *    la función.
 *    IMPORTANTE: Los indices empiezan a contar desde el 0
 * Retorna el numero de operaciones básicas realizadas por metodo;
 * ERR en caso de error.
 **/
int blin_auto(int *tabla,int P,int U,int clave,int *ppos)
{
  int i, cont = 0;

  /* CdE */
  assert(tabla != NULL);
  assert(P >= 0);
  assert(U >= P);
  assert(ppos != NULL);

  for(i = P; i <= U; i++){
    cont ++;
    /* Elemento encontrado */
    if(tabla[i] == clave){
      *ppos = i;

      /* Swap tabla[i] <--> tabla[i-1] */
      if(i != P){
        /* No hace falta variable auxiliar para el swap
         * porque sabemos que tabla[i]=clave */
        tabla[i] = tabla[i-1];
        tabla[i-1] = clave;
      }
      return cont;
    }
  }
  /* Elemento no encontrado */
  *ppos = NO_ENCONTRADO;
  return cont;
}
```

4.2 Apartado 2

```c
/* Autores: Eduardo Terres y Victor Perez
 * Fecha: 4/12/2020
 * Descripcion: Genera un fichero con datos asociados a la ejecucion de un algoritmo de busqueda.
 * Se trata de la funcion principal, que llama a la creacion y busqueda de los datos,
 * asi como su almacenamiento.
 * Parametros entrada:
 * pfunc_busqueda metodo: metodo empleado para la busqueda
 * pfunc_generador_claves generador: funcion generadora de claves a buscar
 * int orden: tipo de tabla - ORDENADA o NO_ORDENADA -
 * int n_veces: numero de veces que busca las claves
 * int N: tamaño de la tabla de datos
 * PTIEMPO ptiempo: estructura que almacena los resultados
 * Retorno:
 * ERR en caso de error; OK en caso contrario */
short tiempo_medio_busqueda(pfunc_busqueda metodo, pfunc_generador_claves generador,
                            int orden, int N, int n_veces, PTIEMPO ptiempo) {
    int i, ob, ppos, ob_min, ob_max;
    long ob_suma;
    PDICC dicc = NULL;
    int *perm = NULL, *claves = NULL;
    clock_t ini_t, fin_t;
    /* Comprobaciones que realiza genera_tiempos_busqueda */
    assert(metodo != NULL);
    assert(generador != NULL);
    assert(ptiempo != NULL);
    assert(n_veces > 0);
    /* Comprobaciones que realiza ini_diccionario */
    assert(orden == ORDENADO || orden == NO_ORDENADO);
    assert(N > 0);
    dicc = ini_diccionario(N, orden);
    if(dicc == NULL) return ERR;
    perm = genera_perm(N);
    if(perm == NULL) return libera_memoria(dicc, NULL, NULL);
    if(insercion_masiva_diccionario(dicc, perm, N) == ERR) return libera_memoria(dicc, perm, NULL);
    claves = (int *) malloc(sizeof(claves[0]) * N * n_veces);
    if(claves == NULL) return libera_memoria(dicc, perm, NULL);
    generador(claves, n_veces * N, N);

    /* Iteraciones del metodo de busqueda */

    /* Clock init value */
    ini_t = clock();
    if (ini_t == (clock_t) - 1) return libera_memoria(dicc, perm, claves);
    /* Inicializacion de parametros */
    ob = 0;
    ob_min = INT_MAX;
    ob_max = 0;
    ob_suma = 0;
    for(i = 0; i < n_veces*N; i++){
        ob = busca_diccionario(dicc, claves[i], &ppos, metodo);
        if(ob == ERR) return libera_memoria(dicc, perm, claves);
        ob_suma += ob;
        if (ob < ob_min) ob_min = ob;
        else if(ob > ob_max) ob_max = ob;
    }

    /* Clock fin value */
    fin_t = clock();
    if (fin_t == (clock_t) - 1) return libera_memoria(dicc, perm, claves);
    /* Almacena los resultados */
    ptiempo->max_ob = ob_max;
    ptiempo->min_ob = ob_min;
    ptiempo->medio_ob = (double) ob_suma/(N*n_veces);
    ptiempo->N = N;
    ptiempo->n_elems = N*n_veces;
    ptiempo->tiempo = (double) (fin_t - ini_t)*1000 / (n_veces * N * CLOCKS_PER_SEC);

    /* Liberacion de memoria */
    libera_memoria(dicc, perm, claves);
    return OK;
}
```

```c
/*
 * Autores: Eduardo Terres y Victor Perez
 * Fecha: 4/12/2020
 *
 * Descripcion: Genera un fichero con datos asociados a la ejecucion de un algoritmo de busqueda.
 * Se trata de la funcion principal, que llama a la creacion y busqueda de los datos,
 * asi como su almacenamiento.
 *
 * Parametros entrada:
 * pfunc_busqueda metodo: metodo empleado para la busqueda
 * pfunc_generador_claves generador: funcion generadora de claves a buscar
 * int orden: tipo de tabla - ORDENADA o NO_ORDENADA -
 * char *fichero: fichero de salida para datos
 * int num_min: tamaño minimo de la tabla de datos
 * int num_max: tamaño maximo de la tabla de datos
 * int incr: incremento en tamaño de tabla
 * int n_veces: numero de veces que busca las claves
 *
 * Retorno:
 * ERR en caso de error; OK en caso contrario
 **/
short genera_tiempos_busqueda(pfunc_busqueda metodo, pfunc_generador_claves generador,
        int orden, char* fichero, int num_min, int num_max, int incr, int n_veces){

    PTIEMPO ptiempos = NULL;
    int n, i, j;
    /* CdE */
    if (metodo == NULL || generador == NULL || fichero == NULL)
        return ERR;
    assert(n_veces > 0);
    assert(incr > 0);
    assert(num_min >= 0);
    assert(num_max > num_min);
    assert(orden == ORDENADO || orden == NO_ORDENADO);
    /* Reajuste de num_max */
    num_max = num_max - (num_max - num_min) % incr;

    n = (num_max - num_min) / incr + 1;
    ptiempos = (PTIEMPO) malloc(n * sizeof(ptiempos[0]));
    if(ptiempos == NULL) return ERR;

    for(i = num_min, j = 0; i <= num_max && j < n; i += incr, j++){
        if( tiempo_medio_busqueda(metodo, generador,
            orden, i, n_veces, &ptiempos[j]) == ERR ){
            free(ptiempos);
            return ERR;
        }
    }
    if(guarda_tabla_tiempos(fichero, ptiempos, n) == ERR){
        free(ptiempos);
        return ERR;
    }

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

```c
/*
 * Funcion: guarda_tabla_tiempos    Fecha: 9/10/2020
 * Autores: Eduardo Terrés y Víctor Pérez
 *
 * Descripcion: Función que escribe en un fichero los
 * tiempos de ejecución dados por un array de estructuras
 * de tiempo.
 *
 * Parametros entrada:

 * char* fichero: ruta del fichero a escribir
 * PTIEMPO tiempo: array de estructuras de tiempo
 * int n_tiempos: longitud del array tiempo
 *
 * Retorno:
 * ERR en caso de error; OK en caso contrario
 **/
short guarda_tabla_tiempos(char* fichero, PTIEMPO t
iempo, int n_tiempos) {
    int i;
    FILE *f = NULL;
    f = fopen(fichero, "w");

    if (f == NULL) return ERR;

    for (i = 0; i < n_tiempos; i++) {
        fprintf(f, "%d %f %f %d %d\n", tiempo[i].N,
                tiempo[i].tiempo,
                tiempo[i].medio_ob,
                tiempo[i].min_ob,
                tiempo[i].max_ob);
    }

    if (fclose(f) != 0) return ERR;
    return OK;
}
```

```c
/**
 * Autores: Eduardo Terres y Victor Perez

 * Fecha: 4/12/2020

 *
 * Descripcion: Esta funcion auxiliar libera la
memoria
 * ocupada por los parametros de entrada
 *
 * Parametros entrada:
 *   -PDICC pdicc: diccionario a liberar
 *   -int *perms: array de enteros a liberar
 *   -int *claves: array de enteros a liberar
 * Retorno:
 * ERR
 **/
int libera_memoria(PDICC dicc, int *perms,
 int *claves){
    if(dicc != NULL){
        libera_diccionario(dicc);
    }
    if(perms != NULL){
        free(perms);
    }
    if(claves != NULL){
        free(claves);
    }
    return ERR;
}
```

5. Resultados, Gráficas

5.1 Apartado 1

5.1.1 TAD Diccionario

Al ejecutar el programa *ejercicio0.c*, se obtiene el siguiente resultado, que confirma la creación correcta de diccionarios ordenados y no ordenados:

Al ejecutar el comando: **./ejercicio0 - tamanio 20 - orden 1**

```
Ejecutando ejercicio0.c para comprobar TAD Diccionario
Practica numero 3, apartado 0
Realizada por: Eduardo Terres y Victor Perez
Grupo: 06
Diccionario:
{ 1      2       3       4       5       6       7       8       9       10      11
12       13      14      15      16      17      18      19      20      }
```

Al ejecutar el comando: ./ejercicio0 - tamanio 20 - orden 0

```
Ejecutando ejercicio0.c para comprobar TAD Diccionario
Practica numero 3, apartado 0
Realizada por: Eduardo Terres y Victor Perez
Grupo: 06
Diccionario:
{ 14    6       10      20      1       8       7       12      18      11      5       21
6       15      3       13      17      4       19      9       }
```

5.1.2 Funciones de búsqueda

Al ejecutar el programa *ejercicio1.c* con los tres algoritmos programados, se obtienen los siguientes resultados:

Para la búsqueda lineal, en diccionarios ordenados:

```
Ejecutando ejercicio1.c para comprobar TAD Diccionario
Practica numero 3, apartado 1
Realizada por: Eduardo Terres y Victor Perez
Grupo: 06
Clave 50 encontrada en la posicion 49 en 50 op. basicas
```

Para la búsqueda lineal, en diccionarios no ordenados:

```
Ejecutando ejercicio1.c para comprobar TAD Diccionario
Practica numero 3, apartado 1
Realizada por: Eduardo Terres y Victor Perez
Grupo: 06
Clave 50 encontrada en la posicion 31 en 32 op. basicas
```

Para la búsqueda binaria, en diccionarios ordenados:

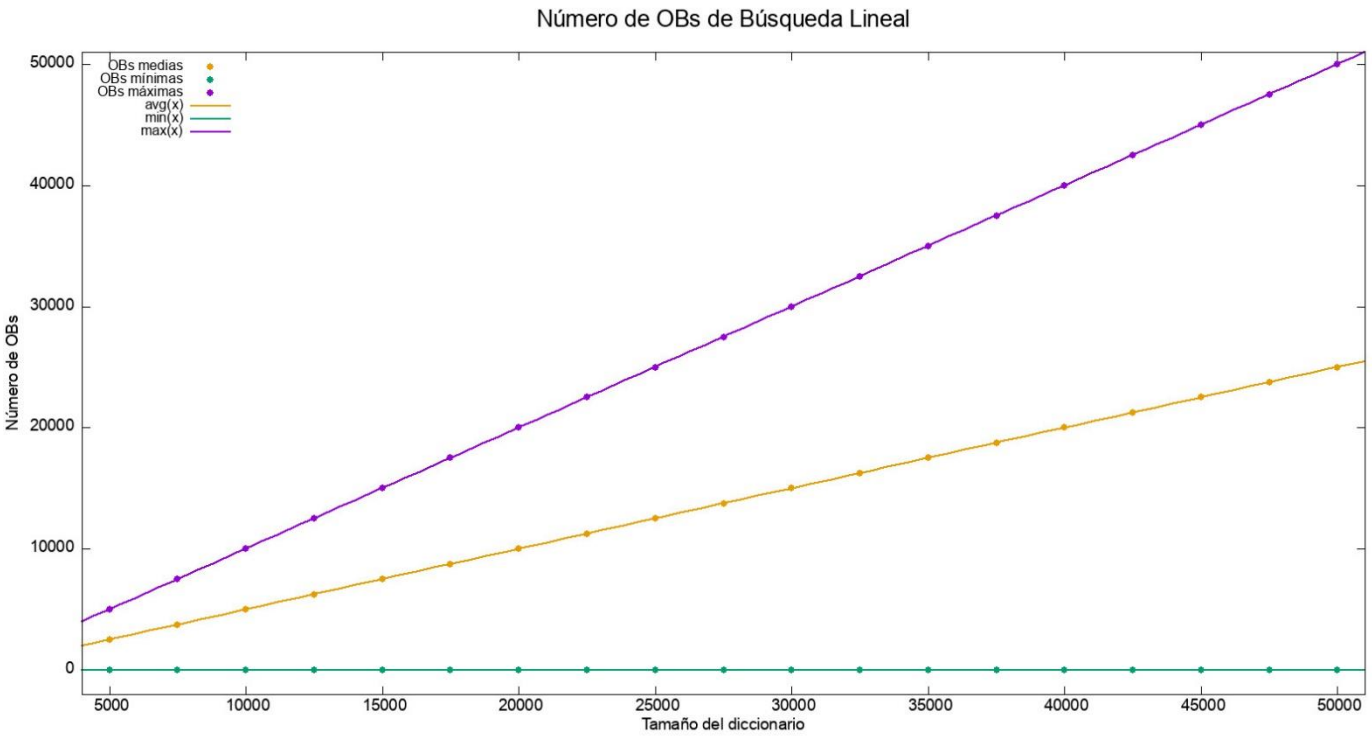
```
Ejecutando ejercicio1.c para comprobar TAD Diccionario
Practica numero 3, apartado 1
Realizada por: Eduardo Terres y Victor Perez
Grupo: 06
Clave 50 encontrada en la posicion 49 en 6 op. basicas
```

Para la búsqueda lineal autoorganizada, el resultado es el mismo que la no autoorganizada – o normal -, ya que solo se ejecuta una búsqueda, por lo que los cambios en la tabla no tienen efecto.

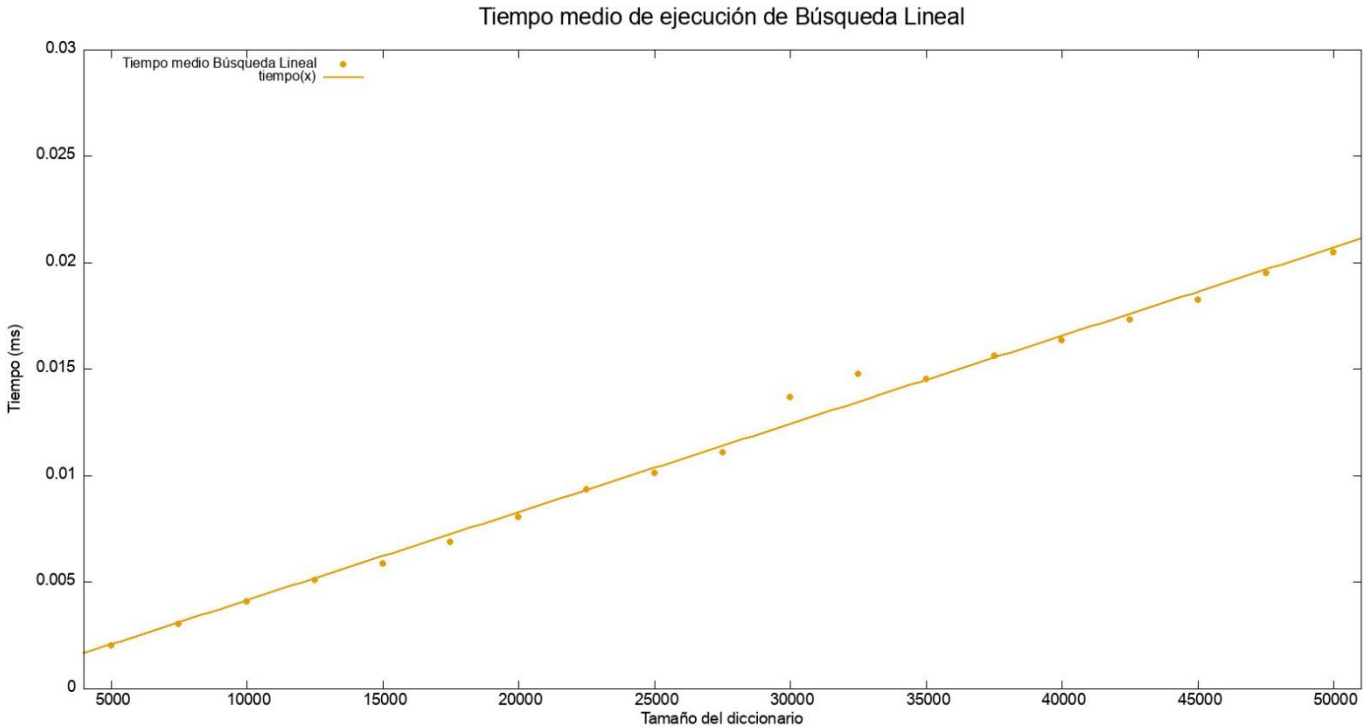
5.2 Apartado 2

Al ejecutar el programa *ejercicio2.c* y dado un algoritmo, se obtienen los ficheros *.dat* que almacenan los datos que se emplearán para representar los resultados. Para comprobar el correcto funcionamiento de este conjunto de funcionalidades, y la correcta programación de las funciones de búsqueda, hemos representado los datos asociados a los siguientes algoritmos:

La siguiente gráfica representa las operaciones básicas para el algoritmo de búsqueda lineal en tablas de tamaño [5000,50.000], con incremento de 2500, generación de claves uniforme y una búsqueda por cada clave en diccionarios no ordenados:

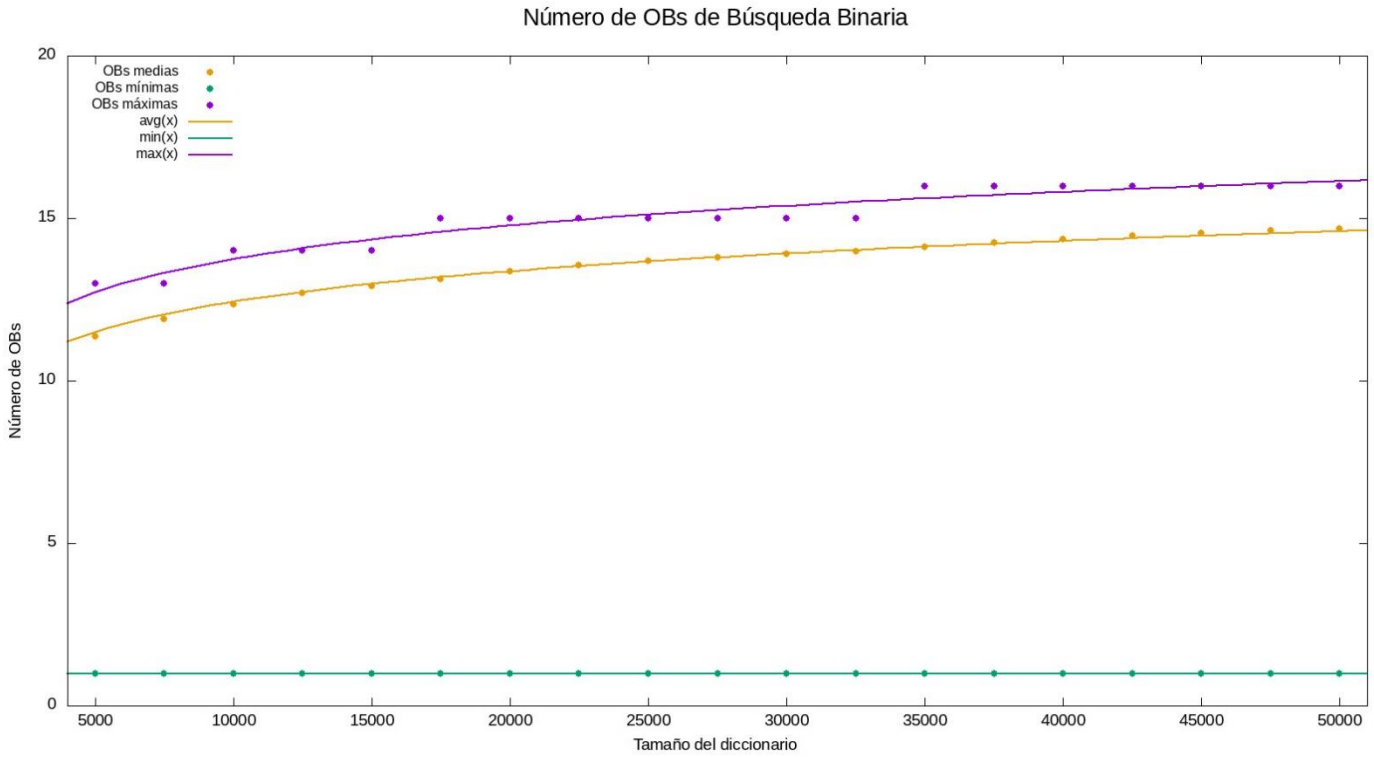


Los datos corresponden con lo esperado a nivel teórico, ya que los crecimientos de las OBs máximas y medias son lineales, siendo el segundo la mitad del primero, y para las mínimas se obtiene la recta $min(x) = 1$, que corresponde al primer elemento de cada tabla. La gráfica del tiempo medio de ejecución es la siguiente:

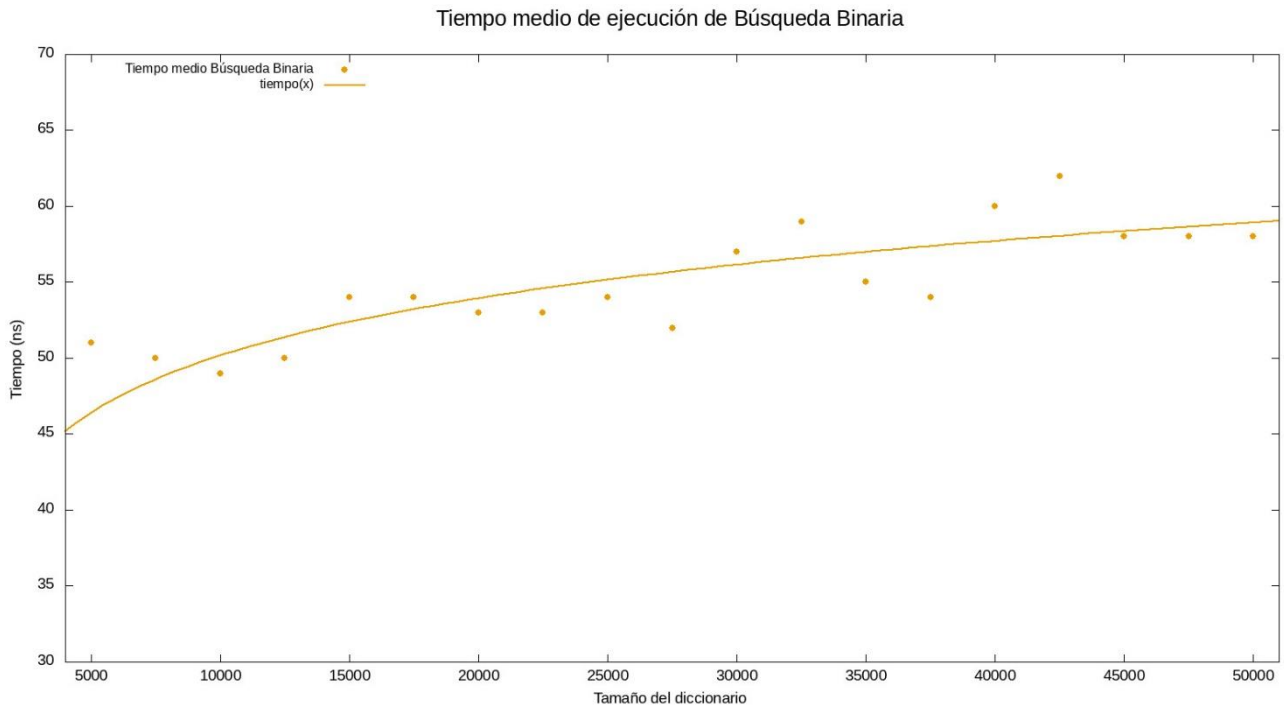


De nuevo, se puede apreciar el crecimiento lineal de este algoritmo y una duración del orden de microsegundos. En comparación a la gráfica anterior, es notable que los puntos obtenidos ya no coinciden con los resultados teóricos, y existen pequeñas fluctuaciones para algunos tamaños. No obstante, el ajuste de curva permite apreciar el crecimiento lineal para el caso de la búsqueda lineal.

Bajo las mismas condiciones de ejecución, a excepción de que se emplea un diccionario ordenado, el algoritmo de búsqueda binaria presenta los siguientes resultados:

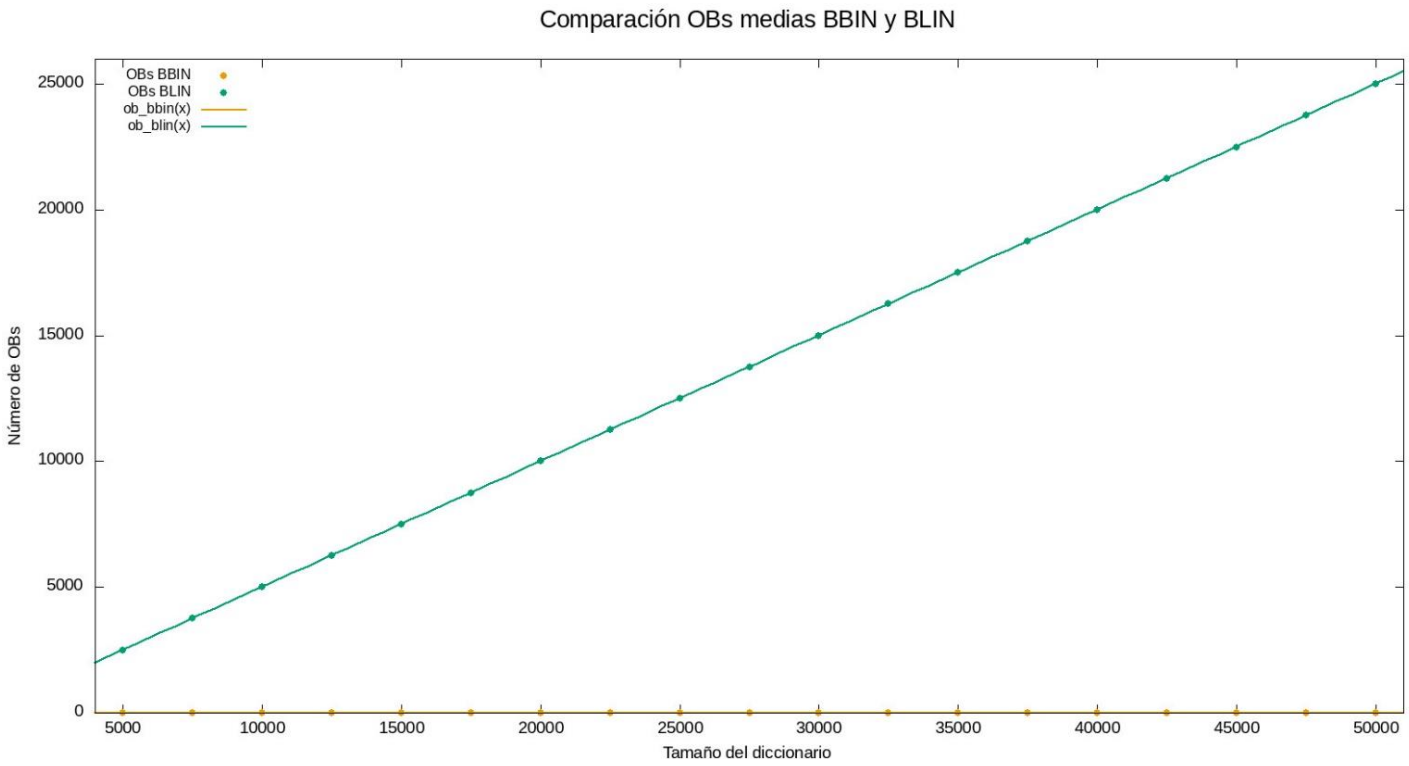


Es notable que el eje de ordenadas tiene como valor máximo 20, lo que hace de este algoritmo un método eficiente para la búsqueda de claves. Los costes temporales máximos y mínimos son cercanos y de carácter logarítmico. Los costes mínimos coinciden con la función $min(x) = 1$, ya que se buscan todas las claves y, en específico, la que se encuentra en la posición $\lfloor (U + P)/2 \rfloor$ tiene coste 1. Al representar el tiempo medio de ejecución, se obtiene el siguiente resultado:

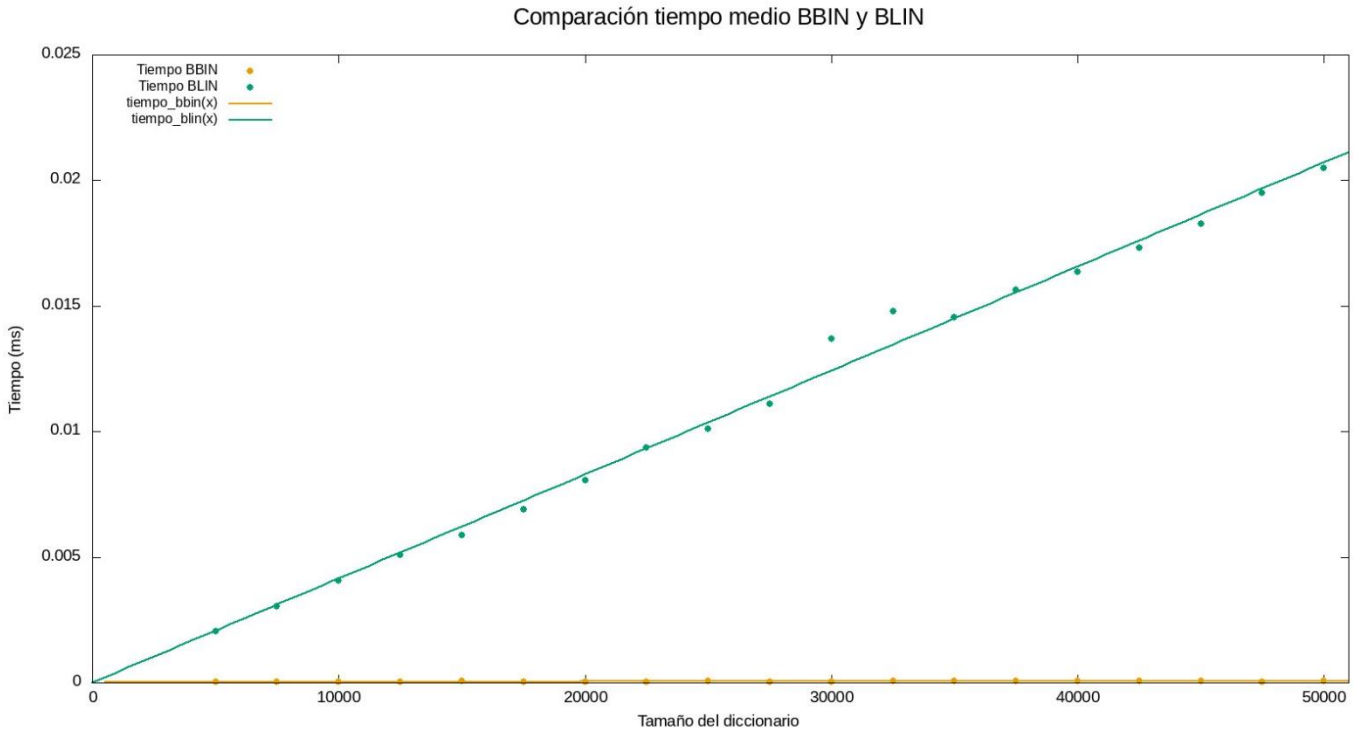


Para un número *n_veces* = 1, los tiempos medios son tan bajos, del orden de nanosegundos, que otros procesos asociados a la ejecución del programa *ejercicio2.c* tienen un importante peso. De todas formas, el crecimiento logarítmico se puede ver reflejado al hacer el ajuste de curva.

A continuación, se compararán los costes medios de los algoritmos previamente presentados:



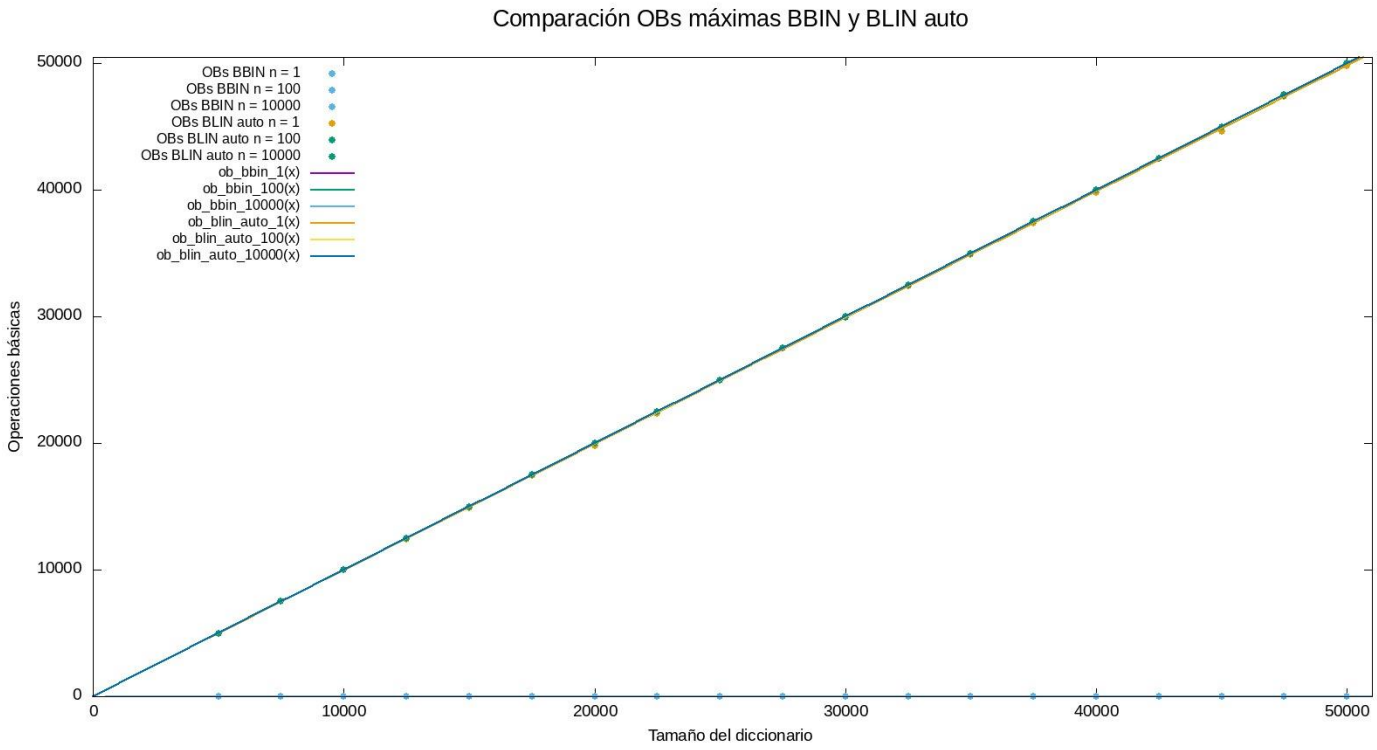
La gráfica anterior permite comparar las operaciones básicas medias que emplea cada algoritmo para buscar claves generadas de forma uniforme en un diccionario de tamaño en el intervalo [5000, 50.000], con incremento de 2500, e iterando 1 vez la búsqueda de cada clave. Gracias a este tipo de generación de las claves, se puede apreciar que los valores obtenidos son exactamente iguales a los teóricos. Se puede apreciar que la diferencia entre ambos métodos es significativa, de forma que el coste temporal medio de la búsqueda binaria – en amarillo – es despreciable respecto a la lineal – en verde -.



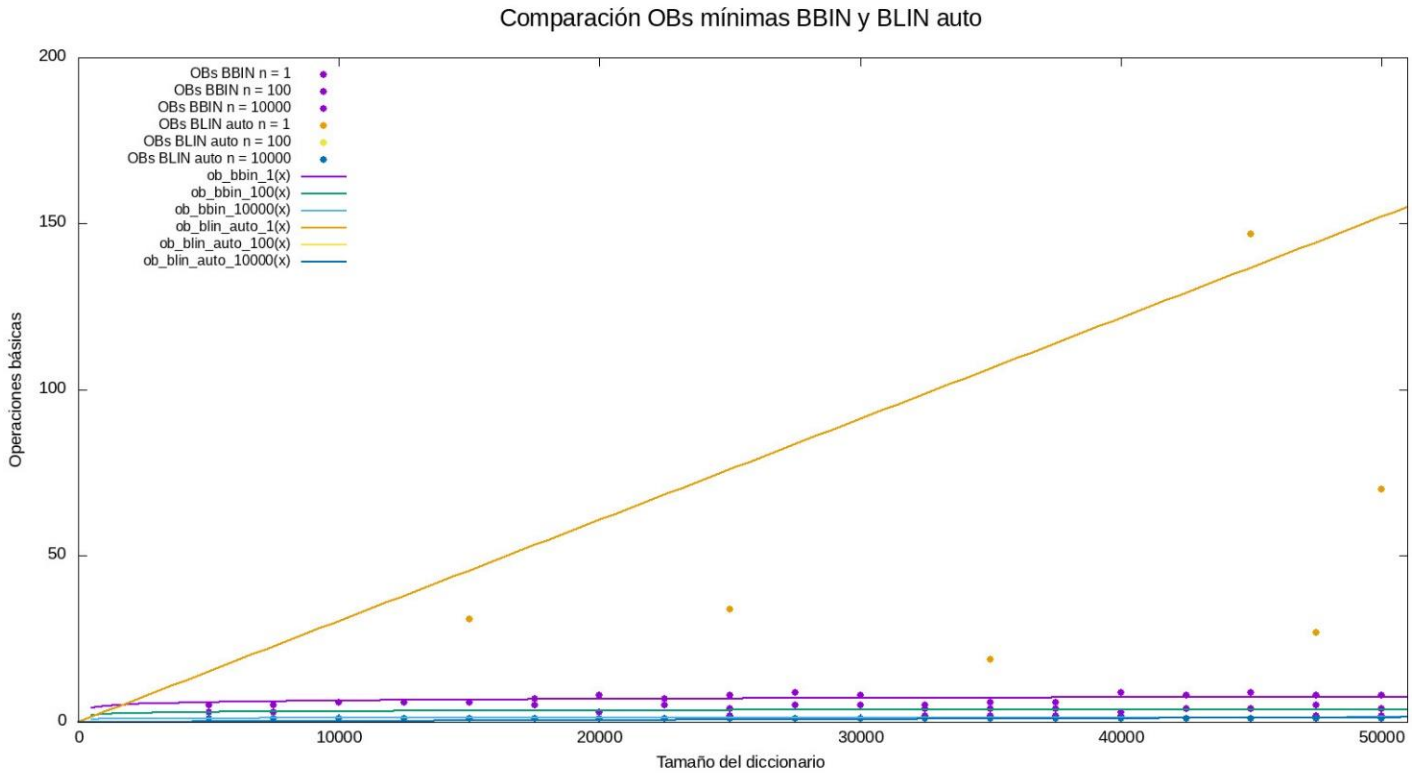
De forma análoga y con los mismos tamaños de tabla y 1 repetición, la gráfica anterior permite comparar el tiempo medio empleado por los algoritmos de búsqueda lineal y binaria. De nuevo, se pueden apreciar las diferencias evidentes entre ambas gráficas, de forma que el tiempo medio para la búsqueda binaria es del orden de nanosegundos, mientras que los costes de la lineal son del orden de microsegundos.

A continuación, se compararán las rutinas de búsqueda lineal, para diccionarios ordenados, y de la búsqueda lineal para diccionarios no ordenados. La generación de claves será de tipo potencial y se repetirá el proceso para n_veces $\in$ {1, 100, 10000}.

Las siguientes gráficas permiten comparar las OBs máximas y mínimas para ambas rutinas:

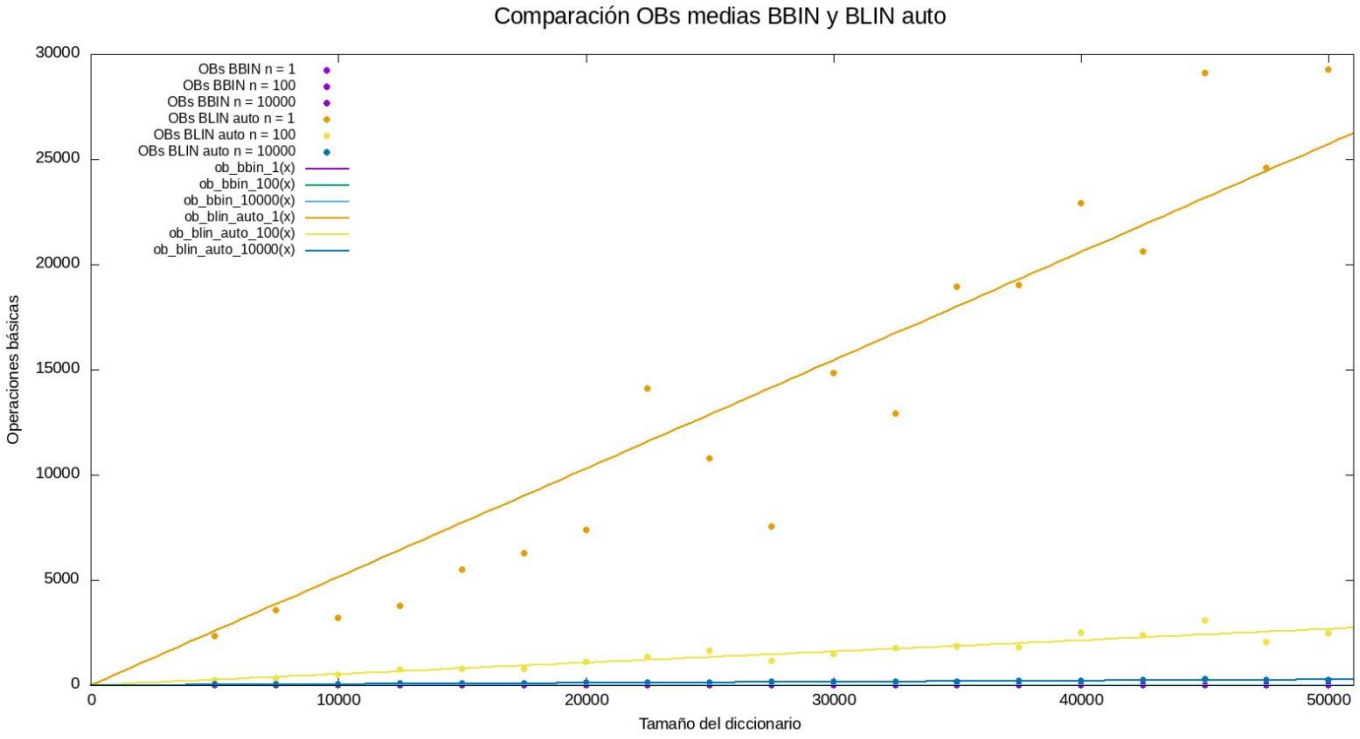


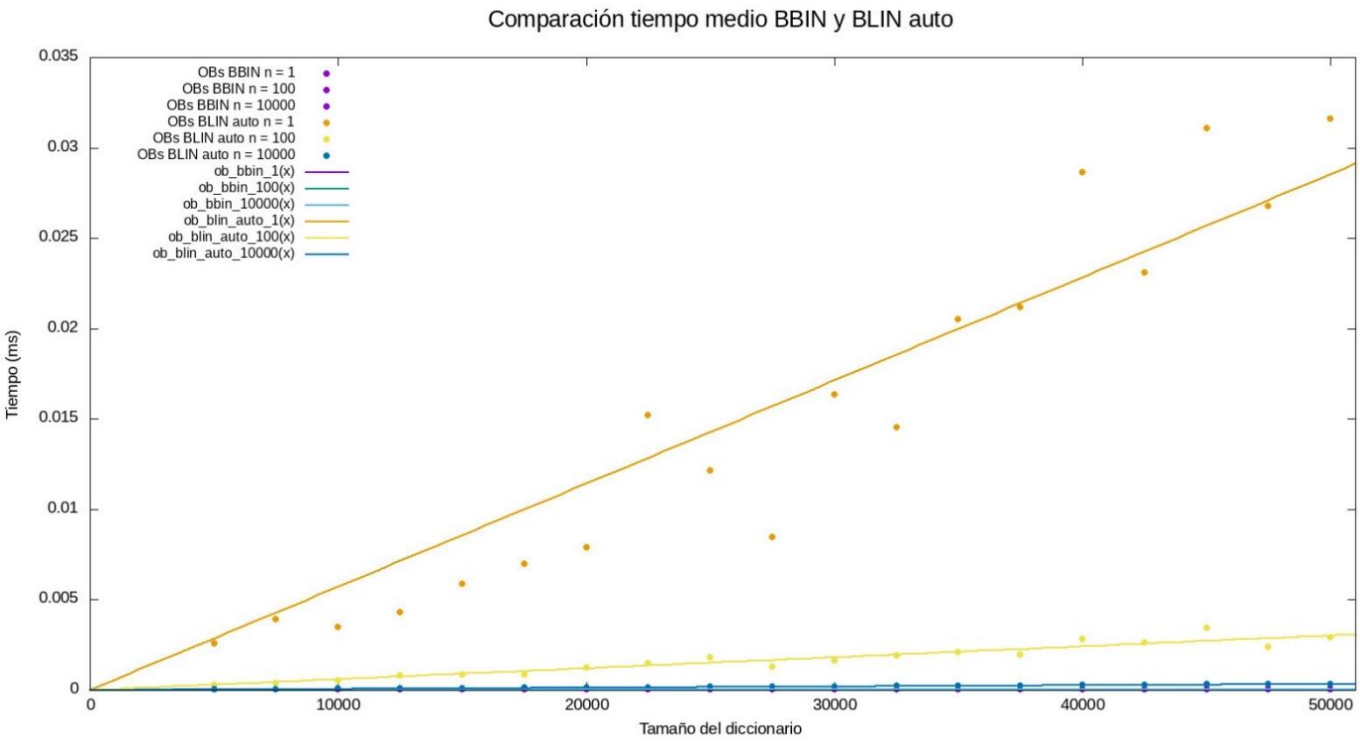
Como es de esperar, y teniendo en cuenta que los resultados de las operaciones básicas máximas quedan representados únicamente por un valor – el máximo del conjunto total de operaciones básicas para un tamaño determinado -, las tres gráficas coinciden prácticamente en una cantidad de OBs igual al tamaño del diccionario. Aún para ejecuciones con n_veces altos, basta con tratar de buscar la clave menos concurrida y, por ende, posicionada al final de la tabla, para obtener un valor cercano al máximo. En el caso de la búsqueda binaria, los costes temporales resultan, de nuevo, despreciables y muy cercanos entre ellos – difieren en pocas unidades para el mismo tamaño -.



La gráfica de las operaciones mínimas presenta diferencias más importantes, en función de las veces que se repite la búsqueda de cada clave del conjunto de claves generado. Es así que, como se trata de diccionarios no ordenados, para una cantidad de *n_veces* baja, los costes mínimos son altos. No obstante, para las ejecuciones de *n_veces* ∈ {100,10000} todas las OBs mínimas han sido 1, o de forma más general, $O(1)$ para búsqueda no uniforme.

Los resultados más interesantes respecto a la evolución del algoritmo de búsqueda lineal autoorganizada se dan en los costes medios, tanto en OBs como en tiempo:





En el caso de las operaciones básicas, para una repetición de la búsqueda, se observa un crecimiento lineal, cercano a N/2, para tablas de tamaño N. No obstante, para *n_veces = 100* los efectos de la autoorganización y la premisa de que el 20% de las claves se buscan un 80% de las veces genera unos costes significativamente menores. Para un número de repetición de búsqueda de cada clave de 10.000, el crecimiento se asemeja más al de tipo logarítmico de la búsqueda binaria. Cabe destacar que, aún así, la búsqueda binaria resulta ser mejor en términos generales y para cualquier valor de *n_veces*, aunque necesita una tabla ordenada.

La tabla del tiempo de ejecución medio refleja los mismos resultados: un tiempo de ejecución de la lineal para *n_veces* = 1 del orden de microsegundos, y del orden de nanosegundos para el resto de funciones.

6. Respuesta a las preguntas teóricas

6.1 Pregunta 1

El algoritmo de búsqueda binaria basa su efectividad en comparar la clave a buscar con la clave en la posición media de la subtabla sobre la que se está buscando. Parece intuitivo, por tanto, razonar que esta comparación de clave es la operación básica de *bbin*. Aunque en el pseudocódigo y en la propia implementación se realizan dos comparaciones, en realidad son la misma operación básica, y en consecuencia se toman como una única comparación de clave.

Para los algoritmos de búsqueda lineal, estándar y auto-organizada (que en esencia realizan el mismo procedimiento hasta una vez encontrada la clave), la operación básica también es una comparación de clave. Sin embargo, en este caso la comparación no verifica si la clave a buscar es mayor, igual o menor, que la que se está comparando; únicamente verifica si es igual (si se ha encontrado o no la clave), y en caso contrario continúa comparando con la clave siguiente.

6.2 Pregunta 2

	Búsqueda lineal	Búsqueda binaria
$W(N)$	N	$\lceil lg(N) \rceil = lg(N) + O(1)$
$B(N)$	1	1

El caso mejor para ambos algoritmos de búsqueda tiene el mismo coste: 1. Esto ocurre cuando la clave a buscar se encuentra en la primera posición que cada algoritmo compara, esto es: la primera posición (índice P) para *blin*; y la posición intermedia, o la anterior si hay un número par de elementos en la tabla (índice $\lfloor (P + U) \,/\, 2 \rfloor$) para *bbin*.

El caso peor para *blin*, es aquel en el que la clave a buscar se encuentra al final de la tabla, en la última posición (índice U). En ese caso *blin* recorrería toda la tabla y compararía la clave con sus N elementos hasta el último.

En el caso peor de *bbin*, la clave a buscar se encontraría en una posición tal que el algoritmo dividiría la tabla en mitades hasta llegar a una subtabla de tamaño 1, cuyo elemento es la clave buscada. Teniendo en cuenta que cada vez que se ha dividido la tabla, es porque se ha realizado una comparación de clave, podemos deducir que, como mucho, se realizarán $\lceil lg(N) \rceil$ divisiones y por tanto comparaciones de clave. De ahí el tiempo de ejecución dado para W_{bbin}.

El caso peor para ambos algoritmos es también aquel en el que la clave que se busca no se encuentra en la tabla, en cuyo caso ambos recorrerían la tabla, cada uno según su procedimiento, tal y como se explica anteriormente.

6.3 Pregunta 3

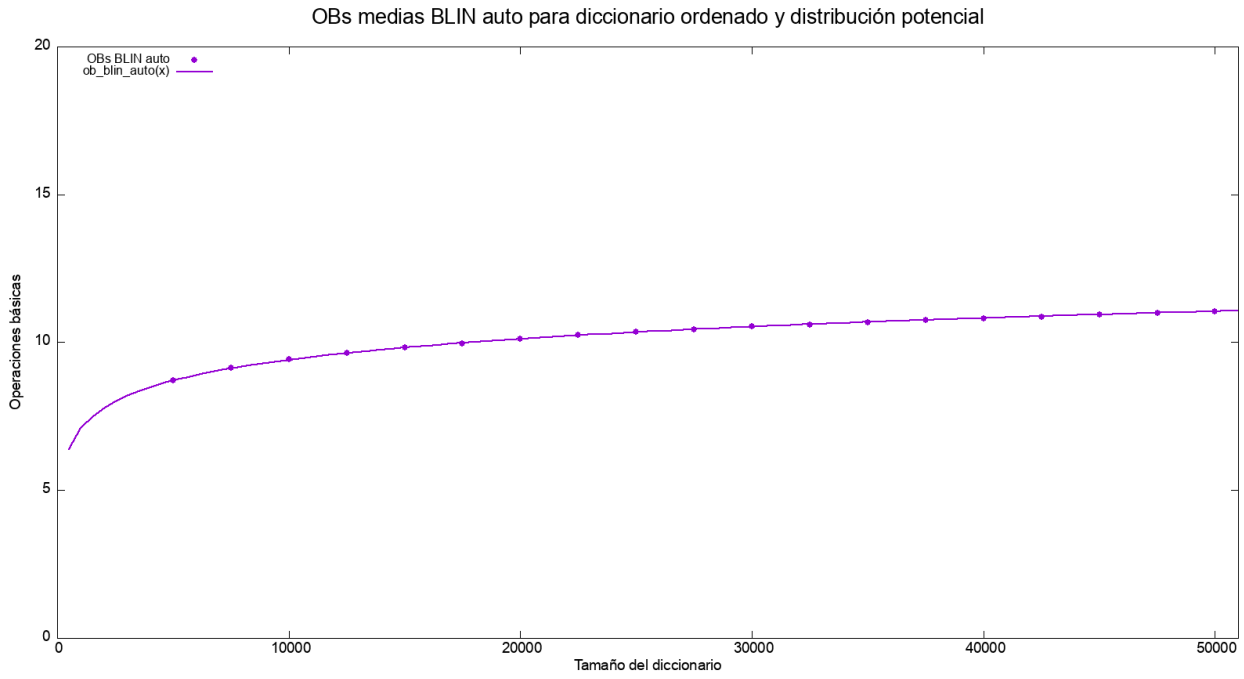
La función *genera_claves_potencial* crea una lista de claves con una distribución no uniforme de tipo potencial, en la que los números más probables son los más pequeños, siendo la probabilidad del 1 del 50%, la del 2 del 17%, etc.

El algoritmo de búsqueda lineal auto-organizada inserta las claves más comunes al principio del diccionario, por lo que podemos concluir que, conforme se va ejecutando el algoritmo, las claves más pequeñas quedan al principio de la tabla. Tras haberse ejecutado un número elevado de veces, el diccionario tendería a quedar ordenado, con unas pocas inversiones puntuales.

6.4 Pregunta 4

Por lo que se ha explicado en la pregunta anterior, tras ejecutar *blin_auto* para una lista de claves de distribución potencial, con un número de claves muy elevado (infinito), en un escenario ideal el diccionario tendería a quedar perfectamente ordenado, pues al ser los números pequeños los más probables, estos quedarían al principio de la tabla. Sobre este escenario, se puede demostrar matemáticamente a partir de la probabilidad de cada clave (dada por la fórmula de *genera_claves_potencial*), que el coste medio de *blin_auto* es de orden logarítmico.

Para comprobar esto, hemos ejecutado *blin_auto* sobre una tabla ordenada (que es como quedaría la tabla idealmente tras haber ejecutado *blin_auto* un número elevado de veces,



pudiendo haber alguna inversión puntual), y hemos generado la tabla de tiempos para una lista de 10000 claves de distribución potencial.

La representación gráfica de esos tiempos, efectivamente, corresponde con una función logarítmica, concretamente, de tipo neperiano.

Suponiendo ahora que el tamaño del diccionario es muy elevado, en una distribución potencial los números más probables siguen siendo unos pocos, para este caso, los más pequeños. Esto nos permite concluir que el algoritmo de búsqueda lineal auto-organizada puede ser más rápido que el de búsqueda binaria para buscar ciertas claves, y con tamaños de diccionario muy elevados y distribuciones de clave potenciales.

6.5 Pregunta 5

El algoritmo de búsqueda binaria posee un importante requisito sin el cuál es totalmente imposible el correcto funcionamiento del algoritmo: la tabla sobre la que se va a buscar debe estar ordenada. Bajo esta condición, el algoritmo *bbin* es capaz de encontrar una clave con un coste de ejecución realmente bajo, de orden logarítmico, incluso para su caso peor.

La explicación de la alta efectividad de este algoritmo se debe a que, en cada paso, la tabla se divide en dos, dejándose de considerar una de las dos mitades. Así, en cada iteración el tamaño de la tabla se va reduciendo por mitades hasta llegar a tablas de un solo elemento, obteniéndose así un tiempo de ejecución logarítmico (de base 2).

$$Paso\ 0: \quad Tam.\ tabla = N$$

$$Paso\ 1: \quad Tam.\ tabla = \frac{N}{2}$$

$$Paso\ 2: \quad Tam.\ tabla = \frac{N}{2^2}$$

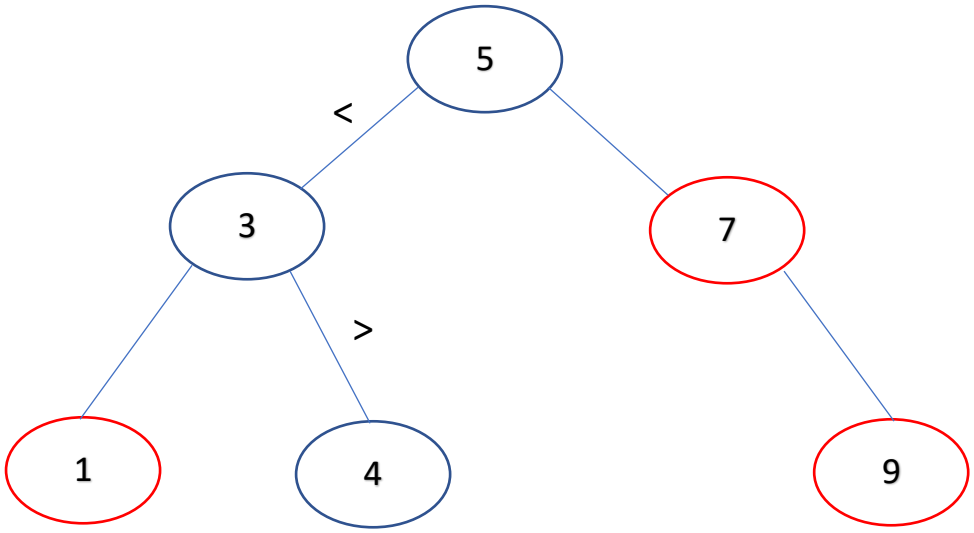
$$\vdots$$

$$Paso\ k: \quad \text{Tam. tabla} = \frac{N}{2^k} \leq 1 \quad \Rightarrow \quad 2^{k-1} < N \leq 2^k \Rightarrow \text{k} = \lceil \lg(\text{N}) \rceil$$

Una forma intuitiva de observar el alto rendimiento de este algoritmo es visualizar la tabla como un árbol binario de búsqueda, localizando en la raíz al elemento medio de la tabla. El primer hijo izquierdo correspondería con el elemento medio de la subtabla izquierda, y el primer hijo derecho con el elemento medio de la subtabla derecha. Así sucesivamente hasta llegar a las hojas que corresponderían con el elemento media de subtablas de tamaño 1.

Estos árboles se denominan así, porque buscar un elemento en ellos es una tarea muy rápida, pues al comparar la clave a buscar con un nodo, se sabe en cuál de sus dos subárboles hijos, va a estar: en el izquierdo si la clave es menor que el nodo, o en el derecho si es mayor. Así, en cada paso se desecha una mitad del árbol y se continúa por el otro subárbol.

Ejemplo: para la tabla [1,3,4,5,7,9] el correspondiente ABdB sería:



(Evolución del ABdB para la búsqueda binaria del 4)

7. Conclusiones finales

Esta última práctica de la asignatura de Análisis de Algoritmos nos ha permitido profundizar en el conocimiento sobre los algoritmos de búsqueda, y la forma de analizarlos y estudiar su rendimiento. Implementar los algoritmos de búsqueda lineal, búsqueda binaria, y búsqueda lineal auto-organizada nos ha ayudado a conocer bien su funcionamiento. Por otra parte, la tarea de analizarlos nos ha hecho partícipes de sus respectivos rendimientos, y de ella hemos podido extraer varias conclusiones.

En general, y siempre que se pueda, el algoritmo de búsqueda binaria va a ser mucho más efectivo que el de búsqueda lineal, pues el primero posee un rendimiento de orden logarítmico mientras que el del segundo es lineal y por tanto superior en tiempo y coste de ejecución. Desafortunadamente, la búsqueda binaria tiene como requisito indispensable que la tabla sobre la que se va a buscar esté ordenada, y esto nos hace plantearnos que, en la mayoría de los casos, no merezca la pena ejecutar un algoritmo de ordenación sobre la tabla para después buscar un elemento en ella, y sea más rentable realizar la búsqueda linealmente.

Hemos observado, que no siempre estaremos buscando claves uniformemente distribuidas y que muchas veces puede darse el caso en el que las claves a buscar se repartan con una distribución potencial. Es en estos casos cuando puede resultar realmente útil el uso de la búsqueda lineal auto-organizada como algoritmo de búsqueda, pues a largo plazo reordenará la tabla haciendo que los elementos más comúnmente buscados sean también los más fáciles de encontrar, situándolos al inicio de la tabla.

Finalmente, hemos visto la utilidad del diccionario como tipo abstracto de datos sobre el que ejecutar estos algoritmos de búsqueda, y la organización de los elementos en él por medio de claves. Además, son particularmente útiles para distinguir entre tablas ordenadas y no ordenadas y, en consecuencia, para saber de antemano qué algoritmo de búsqueda ejecutar sobre ellos.