Análisis de Algoritmos 2019/2020

Práctica 3

César Ramírez & Martín Sánchez, Grupo 1201.

Código	Gráficas	Memoria	Total

1. Introducción.

Esta práctica consistirá en trabajar con algoritmos de búsqueda, estudiar las distintas opciones y obtener conclusiones prácticas sobre los resultados que se obtendrán. Lo primero será crear una estructura llamada *diccionario* la cual contendrá las claves disponibles sobre las que se realizará las distintas búsquedas. A continuación se implementarán tres algoritmos de búsqueda: búsqueda lineal (blin), búsqueda binaria (bbin) y búsqueda lineal auto-organizada (blin_auto).

Con este proyecto se espera obtener una visión más profunda sobre los distintos métodos de búsqueda disponibles, estudiar sus defectos y puntos fuertes, así como aprender el manejo de una estructura diccionario capaz de almacenar claves a buscar.

2. Objetivos

2.1 Apartado 1

En este primer apartado de la práctica lo que hicimos fue crear el TAD (Tipo Abstracto de Datos) Diccionario.

Para ello, primeramente creamos la estructura *diccionario*.

```
typedef struct diccionario {
int tamanio; /* tamaño de la tabla */
int n_datos; /* numero de datos en la tabla */
char orden; /* tabla ordenada (ORDENADO) o desordenada
(NO_ORDENADO) */
int *tabla; /* tabla de datos
} DICC, *PDICC
```

Después creamos todas las funciones relacionadas para implementar este diccionario. En concreto creamos funciones para:

1) Crear un diccionario.
   PDICC ini_diccionario (int tamaño,char orden);

2) Destruir un diccionario.
   void libera_diccionario(PDICC pdicc);

3) Insertar una clave en la posición correcta del diccionario.
   int inserta_diccionario(PDICC pdicc,int clave);

4) Insertar un número grande de claves en la posición correcta del diccionario.
   int insercion_masiva_diccionario (PDICC pdicc, int *claves, int n_claves);

5) Buscar una clave entre todas las posiciones del diccionario.
   int busca_diccionario(PDICC pdicc,int clave,int *ppos,pfunc_busqueda metodo);

Todas las funciones, excepto las de crear y destruir un diccionario, devolverán el número de operaciones básicas, que, como ya sabemos, es justo lo que nos hace falta para calcular y medir el rendimiento de los algoritmos que estudiaremos.

En tercer lugar, implementaremos los algoritmos que ya conocemos de búsqueda binaria y búsqueda lineal. También crearemos un nuevo algoritmo, que es una modificación de la búsqueda lineal. Este se llama búsqueda lineal auto organizada, que, básicamente, lo que hace es cada vez que se busca un elemento en una tabla y este se encuentra, se intercambia el elemento en cuestión con el que está en la posición anterior a este.

De esta manera, a medida que se hacen llamadas a este algoritmo, los elementos más buscados van quedando en las primeras posiciones de la tabla, mientras que los menos buscados quedan al final. Esto hace al algoritmo de búsqueda lineal bastante más eficiente.

Como cabe esperar todas estas funciones devolverán el número de operaciones básicas realizado por estos.

En último lugar hicimos uso del programa ejercicio1.c para comprobar el correcto funcionamiento de las funciones implementadas. En concreto para comprobar el funcionamiento tanto de la búsqueda binaria como de búsqueda lineal sobre una tabla ordenada. Y estas son las evidencias del correcto funcionamiento de los mismos.

-Búsqueda lineal:

```
cesar@cesar-Lenovo-V110-15ISK:~/Documentos/Anal/practica 3/practica 3$ make ejercicio1_blin_test
Ejecutando Ejercicio 1
==6517== Memcheck, a memory error detector
==6517== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==6517== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==6517== Command: ./ejercicio1 -tamanio 20 -clave 16
==6517==
Practica numero 3, apartado 1
Realizada por: Martin Sanchez & Cesar Ramirez
Grupo: 1201
Clave 16 encontrada en la posicion 15 en 16 op. basicas
==6517==
==6517== HEAP SUMMARY:
==6517==     in use at exit: 0 bytes in 0 blocks
==6517==   total heap usage: 4 allocs, 4 frees, 1,208 bytes allocated
==6517==
==6517== All heap blocks were freed -- no leaks are possible
==6517==
==6517== For counts of detected and suppressed errors, rerun with: -v
```

-Búsqueda binaria:

```
cesar@cesar-Lenovo-V110-15ISK:~/Documentos/Anal/práctica 3/práctica 3$ make ejercicio1_bbin_test
Ejecutando Ejercicio 1
==6631== Memcheck, a memory error detector
==6631== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==6631== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==6631== Command: ./ejercicio1 -tamanio 20 -clave 16
==6631==
Practica numero 3, apartado 1
Realizada por: Martin Sanchez & Cesar Ramirez
Grupo: 1201
Clave 16 encontrada en la posicion 15 en 4 op. basicas
==6631==
==6631== HEAP SUMMARY:
==6631==     in use at exit: 0 bytes in 0 blocks
==6631==   total heap usage: 4 allocs, 4 frees, 1,208 bytes allocated
==6631==
==6631== All heap blocks were freed -- no leaks are possible
==6631==
==6631== For counts of detected and suppressed errors, rerun with: -v
```

En ambos tests se pide al algoritmo en cuestión buscar la clave 16 en una tabla ordenada de 20 elementos.

Como se aprecia la búsqueda lineal realiza 16 operaciones básicas, ya que tiene que ir comparando la clave a encontrar con cada elemento de la tabla hasta encontrarla.

En cambio, la búsqueda binaria se deshace de la mitad de posibilidades por cada comparación, quedándose solo con la mitad de la tabla en la que se encuentra la clave con cada comparación.

De ahí las 4 OBs ($\log_2(20) + O(1)$).

2.2 Apartado 2

En este apartado se implementarán una serie de funciones para realizar medidas de rendimiento de las funciones de búsqueda que añadimos a nuestro programa ya existente tiempos.c. Es algo muy parecido a lo que hicimos en las anteriores prácticas con los algoritmos de ordenación, pero esta vez con algoritmos de búsqueda.

La primera función implementada es:

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

La labor de esta función es básicamente es calcular el tiempo medio que ha tardado el algoritmo **metodo** en encontrar una clave en cuestión.

Para esto, la función lo que primero hace es crear un diccionario ordenado o no ordenado, según lo que se le pida a esta.

A continuación crea una permutación de tamaño **N**, que insertaremos en el diccionario gracias a la función **insercion_masiva_diccionario** de búsqueda.c.

Después creamos una tabla que rellenaremos con las claves generadas por el generador en cuestión (**generador**).

Ahora iniciaremos nuestro clock. Tras esto buscaremos en el diccionario las claves de la tabla generada por **generador.** Y por último iniciaremos nuestro segundo clock. Esto nos servirá para calcular el tiempo medio de búsqueda.

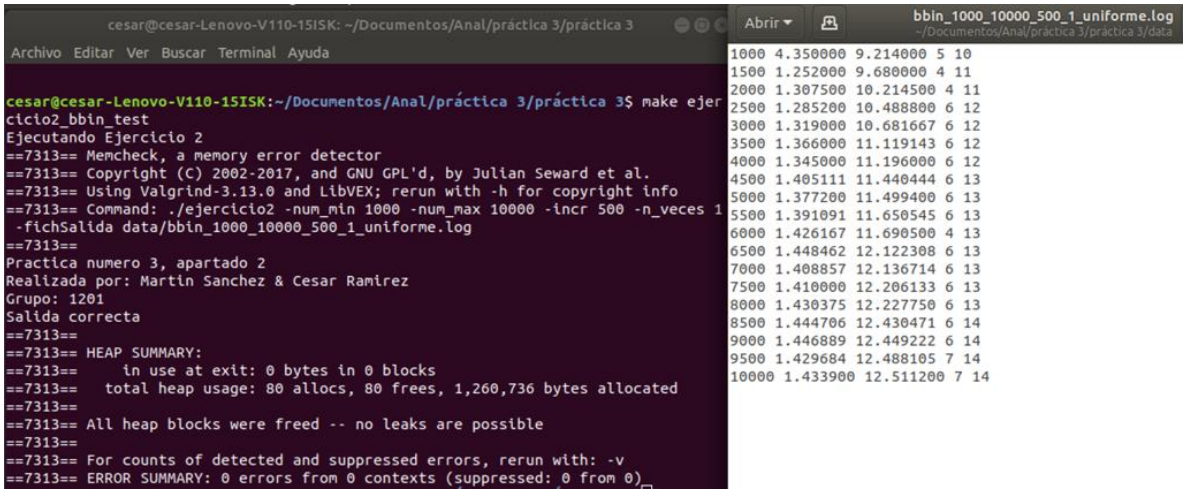
Al final utilizaremos el puntero ptiempo, que es pasado a la propia función, para almacenar en él los datos de: tiempo medio y el número máximo, mínimo y medio de OBs requerido para encontrar las claves.

La segunda y última función implementada es:

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);

Esta función será la encargada de llamar a la función creada antes tantas veces como sea necesario para recopilar toda la información que nos hace falta para medir el rendimiento de los algoritmos implementados. También es la encargada de llamar a la función **guarda_tabla_tiempos,** gracias a la que se generará un fichero .log con toda la información necesaria para analizar el algoritmo en cuestión.

Tras esto, utilizamos el programa ejercicio2.c para comprobar el correcto funcionamiento de las funciones implementadas.



En esta imagen se puede apreciar el correcto funcionamiento de las funciones implementadas. Lo que se ha hecho ha sido realizar el test del algoritmo búsqueda binaria. Para ello se han generado tablas de 1000 a 10000 elementos (ordenados y con un generador de claves uniforme), y búsqueda binaria lo que ha hecho es ir buscando cada clave una sola vez por tabla. Y el resultado con toda la información se ha obtenido en el fichero .log.

Por último, lo que hicimos fue generarnos unos cuantos ficheros .log más para comparar el tiempo medio y el número de OBs medio de los diferentes algoritmos de búsqueda implementados. También construimos una serie de gráficas que más adelante adjuntaremos.

3. Herramientas y metodología

El entorno de desarrollo empleado durante la práctica fue mayoritariamente el disponible en las aulas de laboratorio. En particular, se trabajó principalmente con el sistema operativo de Linux Ubuntu, aunque también se utilizó MacOS cuando los laboratorios no estuvieron disponibles. Por otro lado, el código fue escrito en el editor de texto Atom y compilado mediante Makefile empleando Valgrind. Para las gráficas se utilizó Gnuplot, y finalmente la memoria fue escrita en Microsoft Office Word.

Durante cada apartado la metodología consistió en un desarrollo paralelo de cada una de las funciones, discutiendo las posibles opciones y tomando decisiones conjuntamente, salvo en el caso de las búsquedas, que fueron repartidas y realizadas por separado y luego compartidas.

3.1 Apartado 1

Para la creación de la estructura de diccionario, comenzamos con su definición y cada función a desarrollar fue completándose conjuntamente. Se optó por una inserción basada en insert_sort en caso de que la tabla estuviese ordenada. Una vez finalizadas las primitivas básicas de diccionario, las cuales se realizaron rápidamente, continuaron las búsquedas. Éstas fueron realizadas por separado para luego combinarlas. Una vez finalizado el fichero búsqueda.c se prosiguió al siguiente apartado.

3.2 Apartado 2

Aquí la comparación de tiempos supuso algún que otro inconveniente. Si bien su implementación estaba basada en la ya realizada función para guardar los tiempos de ordenación, en este caso la complejidad era mayor. Una vez finalizadas las funciones, obtuvimos resultados negativos en los tests que hicimos para tamaños relativamente altos, concluyendo que se trataba de un problema de overflow, optamos por modificar las sumas de operaciones básicas. En vez de calcular la media al final, decidimos dividir en cada paso, manteniendo el total en un número manejable. Una vez solventado este incidente, pudimos obtener números medios de operaciones básicas concluyendo así el apartado.

4. Código fuente

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

4.1 Apartado 1

```
PDICC ini_diccionario (int tamanio, char orden) {
  if(tamanio <= 0 || (orden != ORDENADO && orden != NO_ORDENADO)) {
    return NULL;
  }

  PDICC dicc = NULL;
  int * tabla = NULL;
  dicc = (PDICC)malloc(sizeof(DICC));
  if (dicc == NULL){
    return NULL;
  }

  tabla = (int*)malloc(tamanio*sizeof(int));
  if (tabla == NULL){
    free(dicc);
    return NULL;
  }

  dicc->tabla = tabla;
  dicc->tamanio = tamanio;
  dicc->orden = orden;
  dicc->n_datos = 0;

  return dicc;
}


void libera_diccionario(PDICC pdicc)
{
  if (pdicc == NULL){
    return;
  }

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


int inserta_diccionario(PDICC pdicc, int clave)
{
  int i;
  if (pdicc == NULL || (pdicc->orden != ORDENADO && pdicc->orden != NO_ORDENADO) || pdicc->tamanio < (pdicc-
>n_datos + 1)){
    return ERR;
  }
  if (pdicc->orden == NO_ORDENADO){
    pdicc->tabla[pdicc->n_datos] = clave;
    pdicc->n_datos++;

    return OK;
  }

  i = pdicc->n_datos - 1;
  while (i >= 0 && pdicc->tabla[i] > clave){
    pdicc->tabla[i+1] = pdicc->tabla[i];
    i--;
  }
  pdicc->tabla[i+1] = clave;
  pdicc->n_datos++;
```

```c
  return OK;
}


int insercion_masiva_diccionario (PDICC pdicc,int *claves, int n_claves)
{
  int i;

  if( pdicc == NULL || claves == NULL || n_claves < 1){
    return ERR;
  }

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

  return OK;
}


int busca_diccionario(PDICC pdicc, int clave, int *ppos, pfunc_busqueda metodo)
{
  if (pdicc == NULL || ppos == NULL || metodo == NULL || pdicc->n_datos <= 0){
    return ERR;
  }
  return metodo(pdicc->tabla, 0, pdicc->n_datos-1, clave, ppos); /* ppos porque metodo ya te lo hace *ppos*/
}



/* Funciones de busqueda del TAD Diccionario */
int bbin(int *tabla,int P,int U, int clave, int *ppos)
{
  if (tabla == NULL || U < P || ppos == NULL || clave < 0){
    return ERR;
  }

  int m = 0, obs = 0;

  while (P <= U) {
    obs ++;
    m = (P + U)/2;
    if (tabla[m] == clave){
      *ppos = m;
      return obs;
    }
    else if (clave < tabla[m]){
      U = m - 1;
    }
    else{
      P = m + 1;
    }
  }
    *ppos = NO_ENCONTRADO;
    return obs;
}
```

```c
int blin(int *tabla,int P,int U,int clave,int *ppos)
{
  if (tabla == NULL || U < P || ppos == NULL || clave < 0){
    return ERR;
  }

  int i = P, obs = 0;

  while (i <= U){
    obs ++;
    if(tabla[i] == clave){
      *ppos = i;
      return obs;
    }
    i++;
  }

  *ppos = NO_ENCONTRADO;
  return obs;
}
```

```c
int blin_auto(int *tabla,int P,int U,int clave,int *ppos)
{
  if (tabla == NULL || U < P || ppos == NULL || clave < 0){
    return ERR;
  }

  int i = P, obs = 0;

  while (i <= U){
    obs ++;
    if(tabla[i] == clave){
      if(i > P){
        SWAP(tabla[i], tabla[i-1]);
        i--;
      }
      *ppos = i;
      return obs;
    }
    i++;
  }

  *ppos = NO_ENCONTRADO;
  return obs;
}
```

4.2 Apartado 2

```c
short tiempo_medio_busqueda(pfunc_busqueda metodo, pfunc_generador_claves generador,
                            int orden,
                            int N,
                            int n_veces,
```

```
                                PTIEMPO ptiempo){
  if(metodo == NULL || generador == NULL  || (orden != NO_ORDENADO && orden != ORDENADO) || n_veces < 1 ||
ptiempo == NULL || N < 1){
    return ERR;
  }

  clock_t t1;
  clock_t t2;
  int * perm = NULL;
  int pos = 0;
  PDICC pdicc = NULL;
  int* claves = NULL;
  int j = 0;
  double obs_medio = 0;
  int obs_min = INT_MAX;
  int obs_max = -1;
  int obs = 0;
  int n_claves = N*n_veces;

  ptiempo->N = N;
  ptiempo->n_elems = n_claves;

  pdicc = ini_diccionario(N, orden);
  if (pdicc == NULL){
    return ERR;
  }

  perm = genera_perm(N);
  if (perm == NULL){
    libera_diccionario(pdicc);
    return ERR;
  }
  if (insercion_masiva_diccionario(pdicc,perm, N) == ERR){
    libera_diccionario(pdicc);
    free(perm);
    return ERR;
  }

  free(perm);

  claves = (int*)malloc(n_claves*sizeof(int));
  if(claves == NULL){
    libera_diccionario(pdicc);
    return ERR;
  }

  generador(claves, n_claves, N);

  t1=clock();
  if(t1 == ERR) {
    libera_diccionario(pdicc);
    free(claves);
    return ERR;
  }

  for(j = 0; j < n_claves; j++){
    obs = busca_diccionario(pdicc, claves[j], &pos , metodo);
    if(obs == ERR){
      libera_diccionario(pdicc);
      free(claves);
```

```
      return ERR;
    }

    if(obs < obs_min) {
      obs_min = obs;
    }
    else if(obs > obs_max) {
      obs_max = obs;
    }
    obs_medio += (double)obs/n_claves;
  }

 t2=clock();
 if(t2 == ERR) {
   libera_diccionario(pdicc);
   free(claves);
   return ERR;
 }

 ptiempo->tiempo = (double)(t2 - t1)/n_claves;
 ptiempo->medio_ob = obs_medio;
 ptiempo->min_ob = obs_min;
 ptiempo->max_ob = obs_max;

 libera_diccionario(pdicc);
 free(claves);

 return OK;

}



short genera_tiempos_busqueda(pfunc_busqueda metodo, pfunc_generador_claves generador,
                              int orden, char* fichero,
                              int num_min, int num_max,
                              int incr, int n_veces){
  if(metodo == NULL || generador == NULL  || (orden != NO_ORDENADO && orden != ORDENADO) || n_veces < 1 ||
fichero == NULL || num_min < 0 || num_max < num_min || incr < 1){
    return ERR;
  }
  int size = (num_max - num_min)/incr + 1;  /* Numero de tiempos a medir*/
  int j = 0;
  int N = num_min;
  PTIEMPO tiempos = NULL;

  tiempos = (PTIEMPO)malloc(size*sizeof(TIEMPO));
  if (tiempos == NULL) return ERR;

  for(j=0; j<size; j++) {
  /* Guardamos en tiempos[j] el tiempo medio para ordenar n_perms de tam N */
    if(tiempo_medio_busqueda(metodo, generador, orden, N ,n_veces, &tiempos[j]) == ERR) {
      free(tiempos);
      return ERR;
    }
    N += incr;
  }

  if(guarda_tabla_tiempos(fichero, tiempos, size)==ERR) {
    free(tiempos);
```

```
    return ERR;
  }

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

5. Resultados, Gráficas

5.1 Apartado 1

Vamos a observar el correcto funcionamiento de los tres algoritmos de búsqueda implementados. Para ello les pediremos buscar la clave 27 en una tabla de 50 elementos.

-Búsqueda binaria:

```
cesar@cesar-Lenovo-V110-15ISK:~/Documentos/Anal/práctica 3/práctica 3$ make ejer
cicio1_bbin_test
Ejecutando Ejercicio 1
==7701== Memcheck, a memory error detector
==7701== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==7701== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==7701== Command: ./ejercicio1 -tamanio 50 -clave 27
==7701==
Practica numero 3, apartado 1
Realizada por: Martin Sanchez & Cesar Ramirez
Grupo: 1201
Clave 27 encontrada en la posicion 26 en 6 op. basicas
==7701==
==7701== HEAP SUMMARY:
==7701==     in use at exit: 0 bytes in 0 blocks
==7701==   total heap usage: 4 allocs, 4 frees, 1,448 bytes allocated
==7701==
==7701== All heap blocks were freed -- no leaks are possible
==7701==
==7701== For counts of detected and suppressed errors, rerun with: -v
==7701== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Vemos que la clave encontrada se ha encontrado en la posición 26 y en 6 OBs. ($\log_2(50)$ +O(1).

-Búsqueda lineal:

```
cesar@cesar-Lenovo-V110-15ISK:~/Documentos/Anal/práctica 3/práctica 3$ make ejer
cicio1_blin_test
Ejecutando Ejercicio 1
==7610== Memcheck, a memory error detector
==7610== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==7610== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==7610== Command: ./ejercicio1 -tamanio 50 -clave 27
==7610==
Practica numero 3, apartado 1
Realizada por: Martin Sanchez & Cesar Ramirez
Grupo: 1201
Clave 27 encontrada en la posicion 26 en 27 op. basicas
==7610==
==7610== HEAP SUMMARY:
==7610==     in use at exit: 0 bytes in 0 blocks
==7610==   total heap usage: 4 allocs, 4 frees, 1,448 bytes allocated
==7610==
==7610== All heap blocks were freed -- no leaks are possible
==7610==
==7610== For counts of detected and suppressed errors, rerun with: -v
==7610== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Su correcto funcionamiento resulta concluyente.

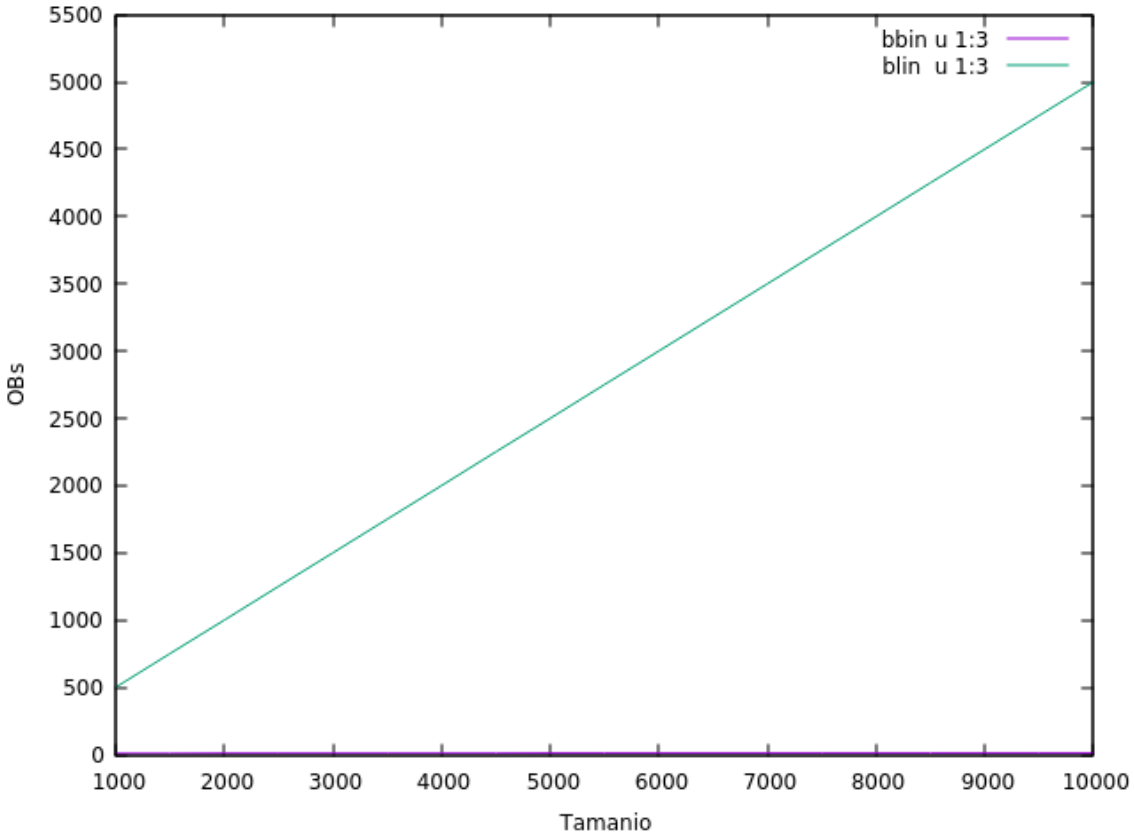
-Búsqueda lineal auto-organizada:

```
cesar@cesar-Lenovo-V110-15ISK:~/Documentos/Anal/práctica 3/práctica 3$ make ejercicio1_blin_auto_te
st
Ejecutando Ejercicio 1
==7956== Memcheck, a memory error detector
==7956== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==7956== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==7956== Command: ./ejercicio1 -tamanio 50 -clave 27
==7956==
Practica numero 3, apartado 1
Realizada por: Martin Sanchez & Cesar Ramirez
Grupo: 1201
Clave 27 encontrada en la posicion 6 en 8 op. basicas
Clave 27 encontrada en la posicion 5 en 7 op. basicas
Clave 27 encontrada en la posicion 4 en 6 op. basicas
Clave 27 encontrada en la posicion 3 en 5 op. basicas
Clave 27 encontrada en la posicion 2 en 4 op. basicas
Clave 27 encontrada en la posicion 1 en 3 op. basicas
Clave 27 encontrada en la posicion 0 en 2 op. basicas
Clave 27 encontrada en la posicion 0 en 1 op. basicas
Clave 27 encontrada en la posicion 0 en 1 op. basicas
==7956==
==7956== HEAP SUMMARY:
==7956==     in use at exit: 0 bytes in 0 blocks
==7956==   total heap usage: 4 allocs, 4 frees, 1,448 bytes allocated
==7956==
==7956== All heap blocks were freed -- no leaks are possible
==7956==
==7956== For counts of detected and suppressed errors, rerun with: -v
==7956== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Para comprobar el correcto funcionamiento de blin_auto se ha modificado ligeramente ejercicio1.c de manera que se llamase varias veces a este método de búsqueda y que, cada vez que se hiciese, se imprimiese en pantalla el resultado. Como sabemos la diferencia entre blin_auto y blin, solo se aprecia cuando se quiere buscar una clave varias veces, donde es el primero mucho más eficiente que el segundo. En esta imagen se puede ver como el elemento 27 (clave a buscar) se va moviendo al principio de la tabla por cada vez que este se busca con blin_auto, hasta quedar al inicio de la misma.

5.2 Apartado 2

Gráfica comparando el número promedio de OBs entre la búsqueda lineal y la búsqueda binaria.



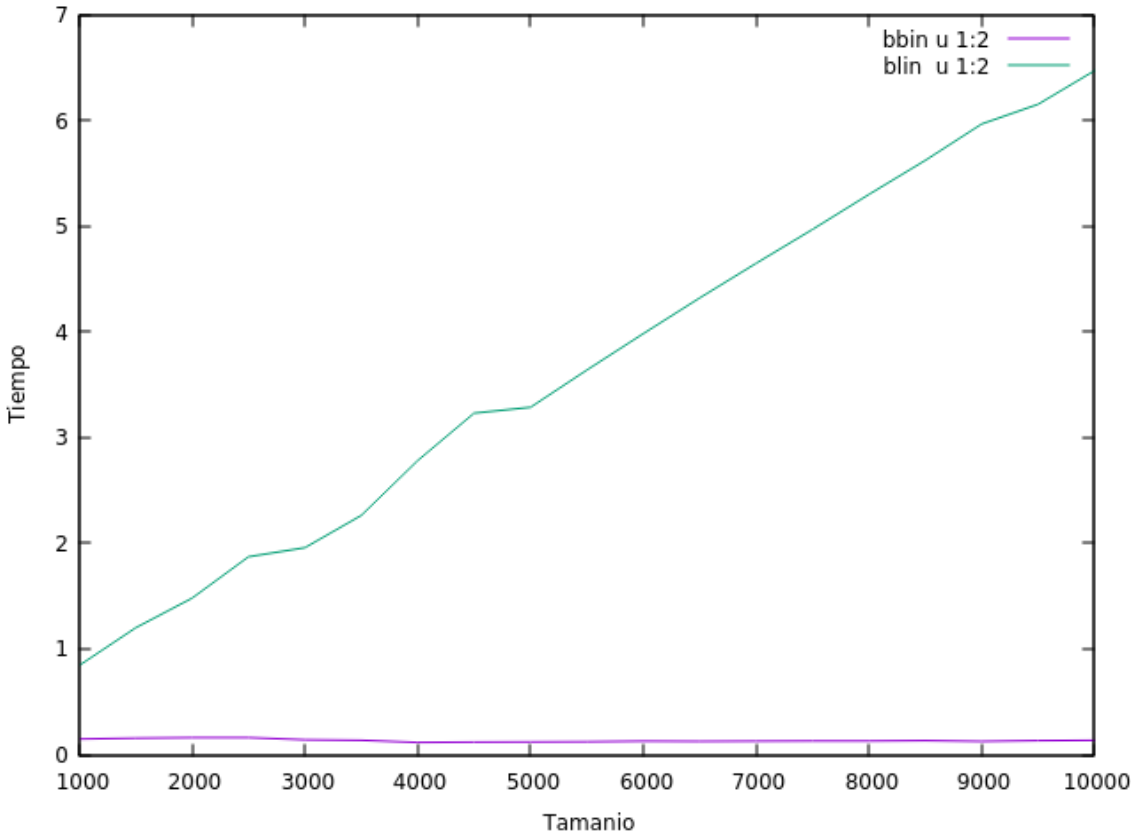
Aquí lo que hicimos fue que bbin buscase, en tablas de 1000 a 10000 elementos, cada elemento 1 sola vez. Y evidentemente con diccionarios ordenados. Y el generador de claves utilizado es el uniforme, que genera claves linealmente.

Sin embargo, blin será aplicado de la misma manera, pero sobre diccionarios no ordenados.

Como es de esperar, el número de OBs en blin, irá aumentando a la vez que lo hace el tamaño de la tabla. Y es evidente que el número de OBs medias es la mitad del tamaño de la tabla. La mitad de los elementos de cada tabla se encontrarán en menos de n/2 OBs y la otra mitad en más. (Donde n es el tamaño de la tabla).

Bbin en cambio se mantiene constante en muy pocas OBs, ya que como todos sabemos el número de OBs realizado por bbin es $\log_2(n) + O(1)$, donde n es el número de elementos de la tabla. Luego el número de OBs oscilará entre $\log_2(1000) \sim 10$ y $\log_2(10000) \sim 13$ aproximadamente.

Gráfica comparando el tiempo promedio de reloj entre la búsqueda lineal y la búsqueda binaria.



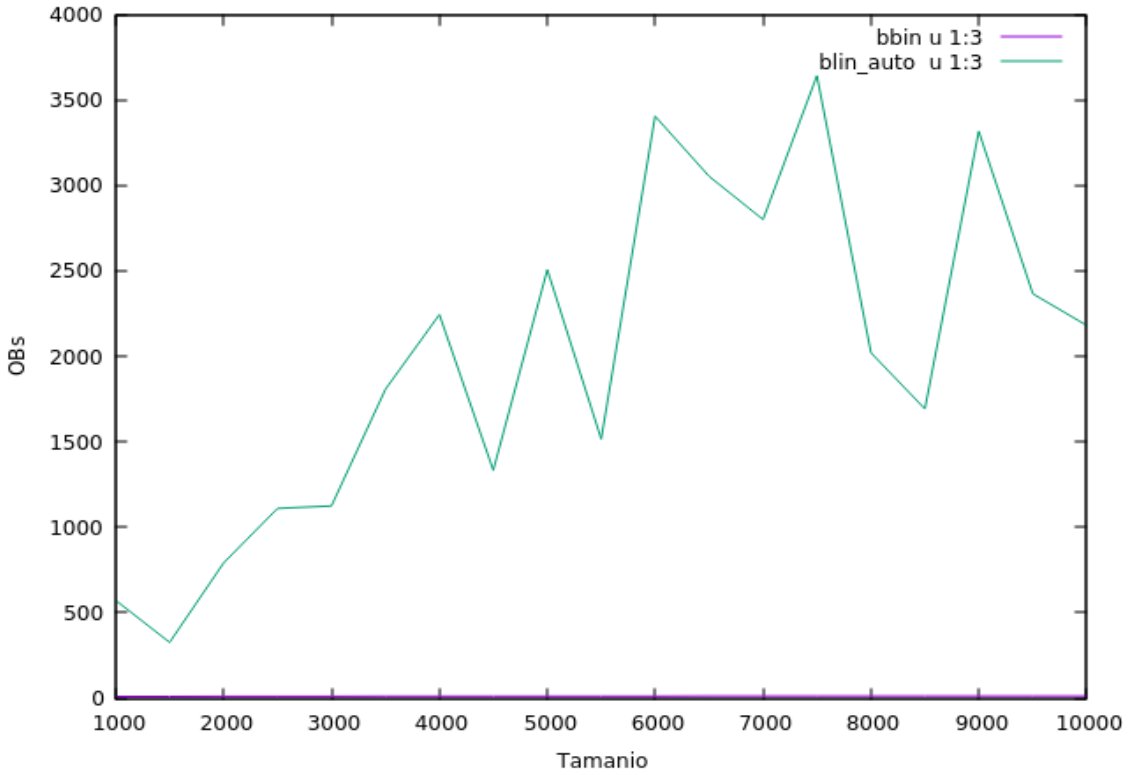
El resultado es parecido al obtenido en la gráfica de OBs. Blin tardará más cuantos más elementos haya, ya que más elementos tendrá que comparar con el elemento buscado hasta encontrarlo.

Y bbin, sin embargo, se mantiene constante, ya que sea la tabla de 1000 o de 10000 elementos, la tabla podrá dividirse por la mitad entre 10 y 13 veces (aprox). Y el algoritmo tarda prácticamente lo mismo en dividir por la mitad una tabla 10 que 13 veces.

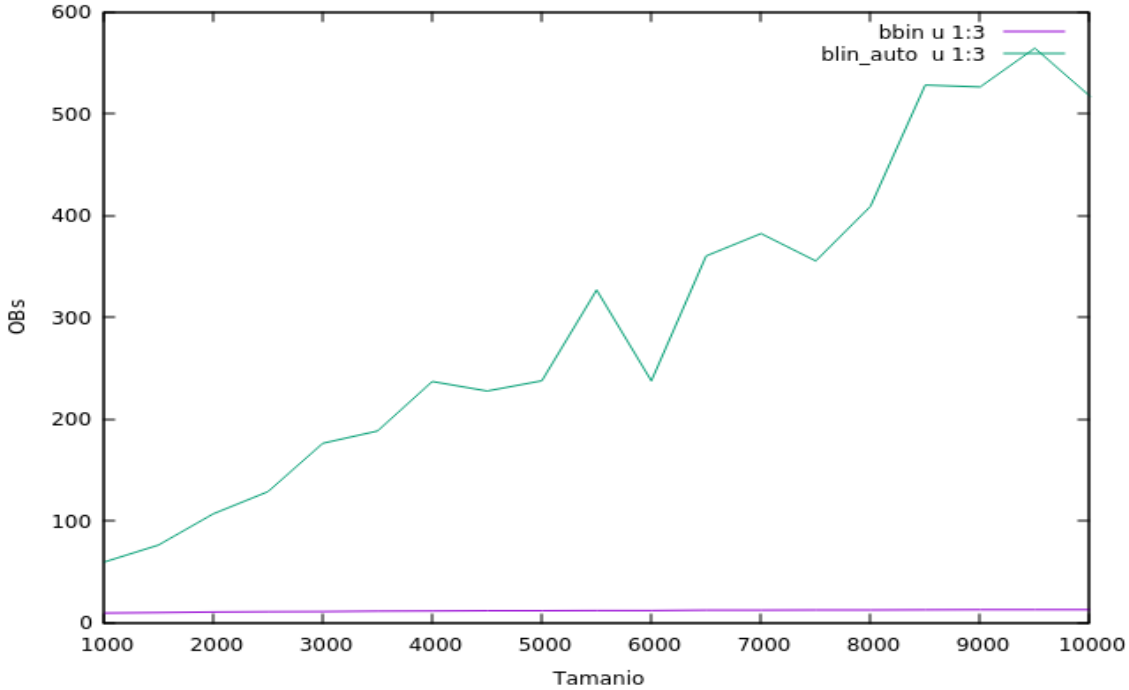
Gráfica comparando el número promedio de OBs entre la búsqueda binaria y la búsqueda lineal auto organizada.

Para realizar estas gráficas utilizamos diccionarios no ordenados para blin_auto y ordenado para bbin, usando el generador de claves potencial.

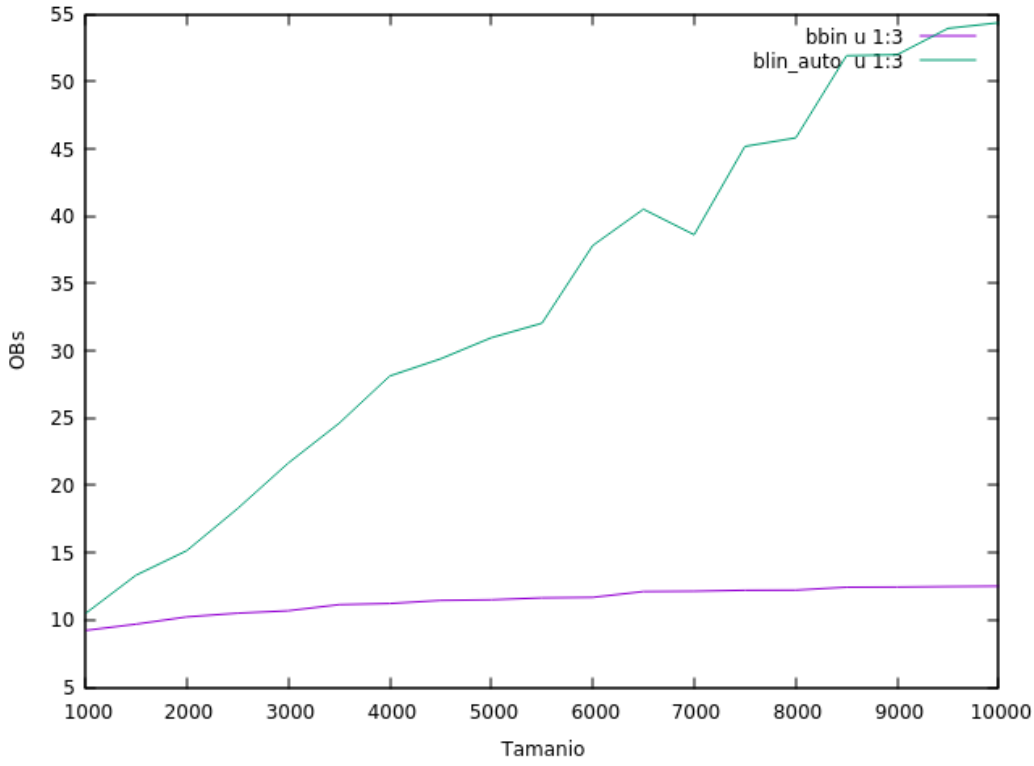
-n_veces = 1



-n_veces = 100



-n_veces = 10000

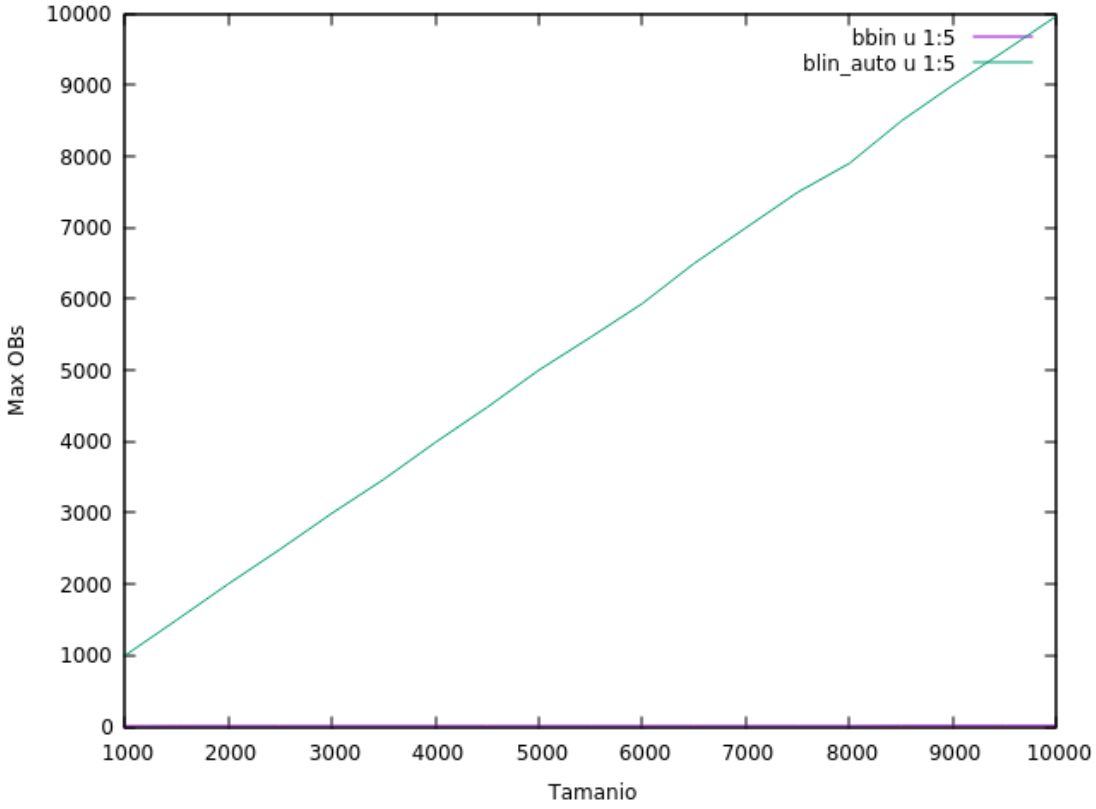


Del número medio de OBs de bbin ya hemos hablado en la gráfica anterior, y aunque en este caso el generador de claves utilizado sea el potencial, los resultados que vamos a obtener van a ser los mismos que antes. Es decir, un número de OBs entre 10 y 13.

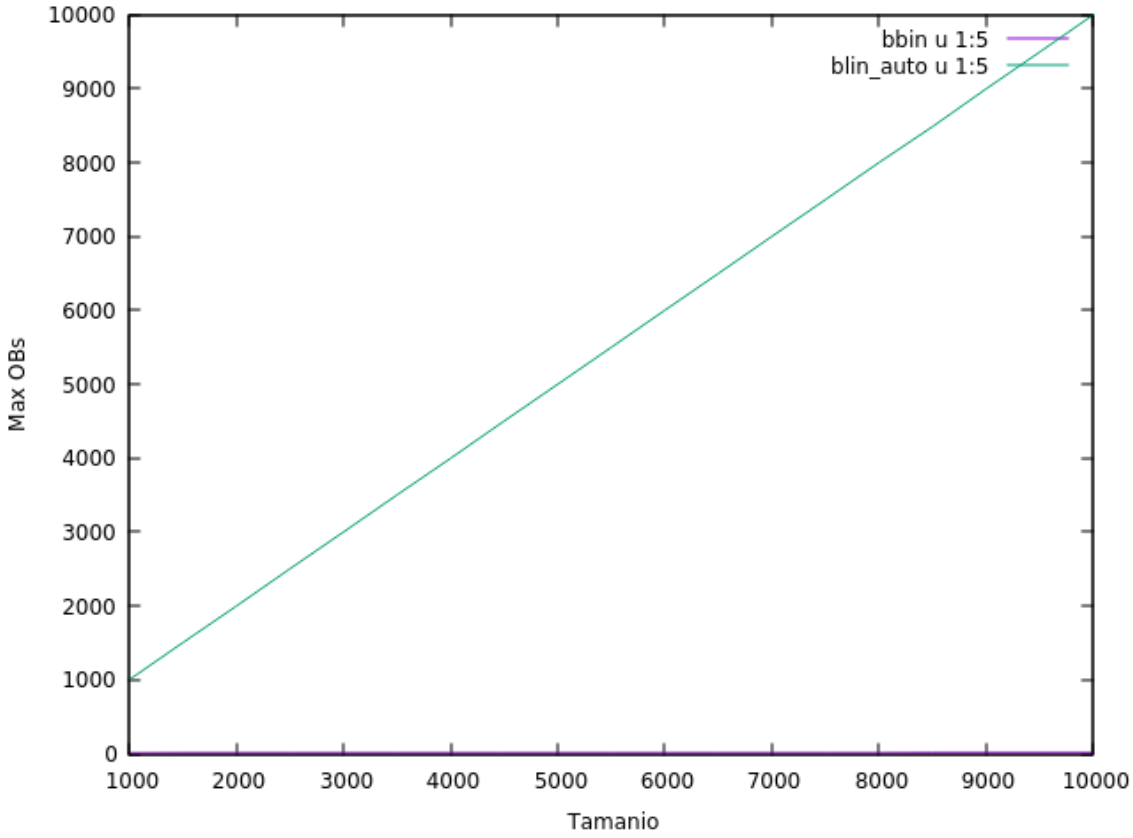
En las diferentes gráficas podemos ver el potencial de blin_auto. Cuantas más veces buscamos cada clave, menos OBs tiene que hacer el algoritmo. Por eso es cuando buscamos 10000 veces cada clave cuando el algoritmo realiza menos OBs de media. Y es cuando buscamos una sola vez cada clave cuando más. Esto es debido al funcionamiento de blin_auto, que cada vez que encuentra la clave buscada en la tabla, la intercambia con el elemento colocado justo en la posición anterior, quedando tras varias llamadas al función los elementos más buscados al inicio de la tabla, y los menos al final. Y esto hace mucho más eficaz al algoritmo.

Gráfica comparando el número máximo de OBs entre la búsqueda binaria y la búsqueda lineal auto organizada.

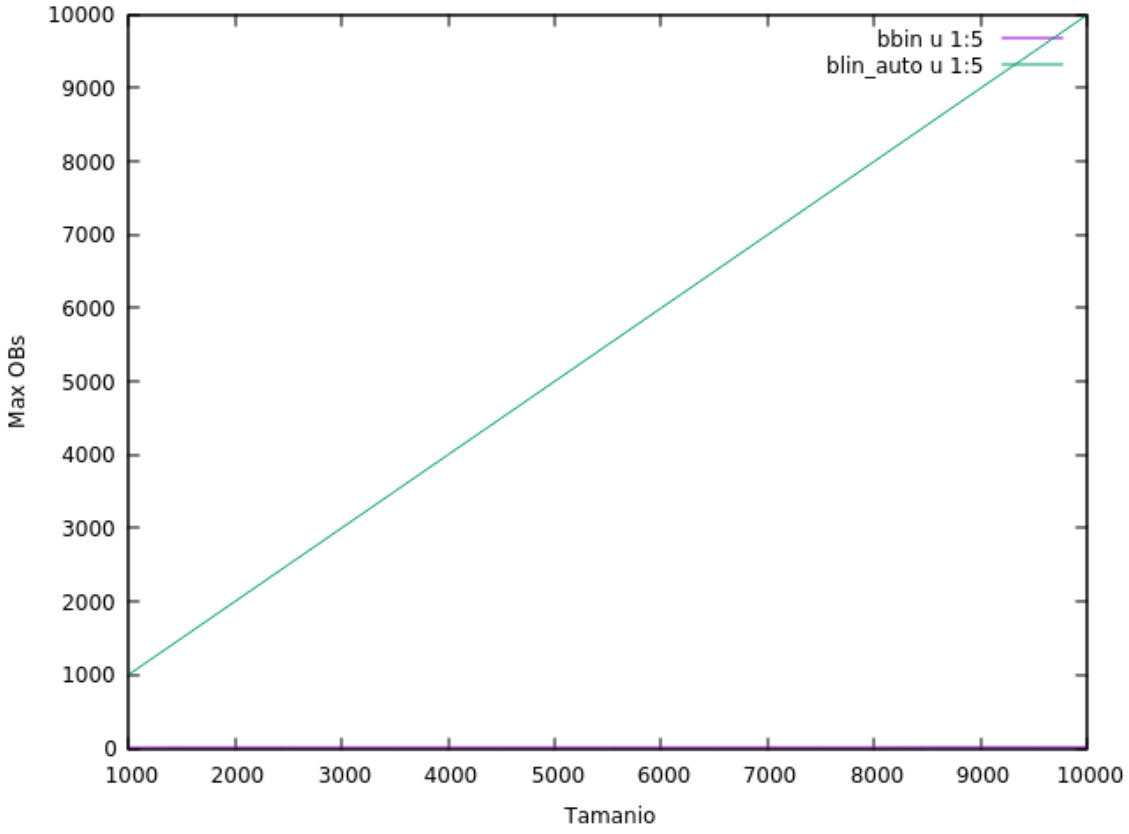
-n_veces = 1



-n_veces = 100



-n_veces = 10000

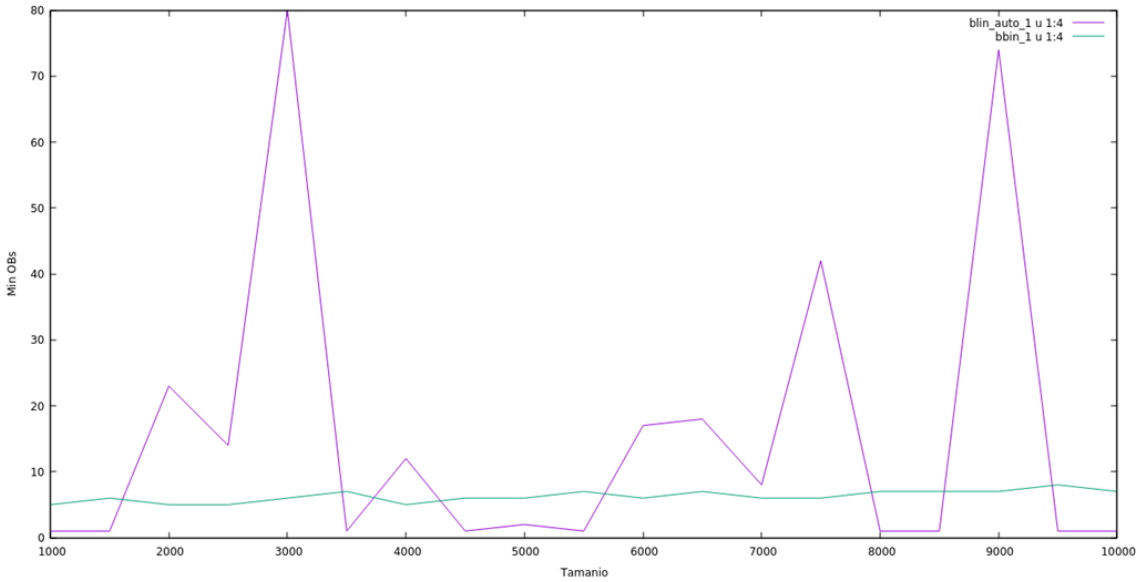


De bbin ya hemos hablado, y sabemos que el número de OBs que realizará va a ser siempre $\log_2(n)$ +O(1). Es decir, es un algoritmo muy eficaz en todos los casos, incluso en el peor. Se podría decir que bbin es un algoritmo de búsqueda óptimo en todos los casos. Siempre con la condición de que la tabla en la que busquemos esté ordenada.

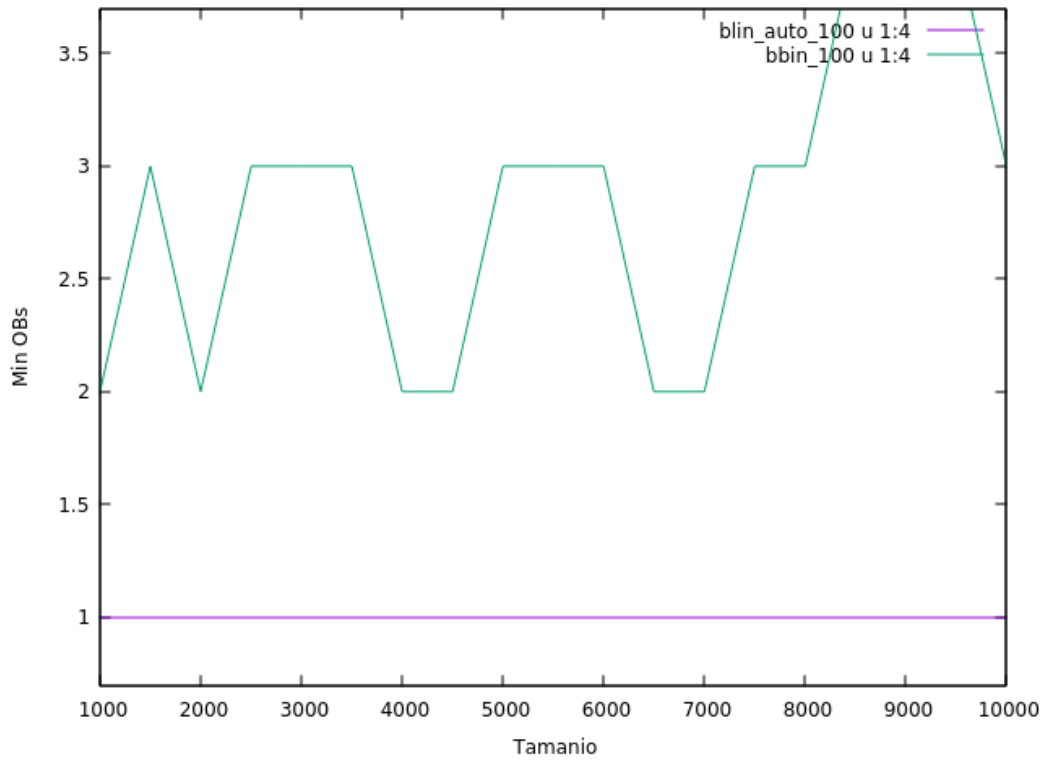
En cambio, blin_auto es un algoritmo de búsqueda que tiene que realizar un montón de OBs un su caso peor. Como se puede ver, aunque busquemos cada clave un gran número de veces, el número máximo de OBs es casi idéntico a cuando buscamos cada clave una sola vez. Podríamos pensar que esto no tiene sentido, ya que, como ya sabemos, blin_auto es verdaderamente fuerte cuando cada clave es buscada muchas veces, pero si pensamos con detenimiento nos damos cuenta de que el caso en el que más OBs tiene que hacer blin_auto, se da cuando una clave es buscada por primera vez. De esta forma da igual que una clave sea buscada una vez, que 10000 veces, que el caso peor siempre va a ser la primera vez que se busque cada clave, así que el número máximo de OBs coincidirá en ambos casos.

Gráfica comparando el número mínimo de OBs entre la búsqueda binaria y la búsqueda lineal auto organizada.

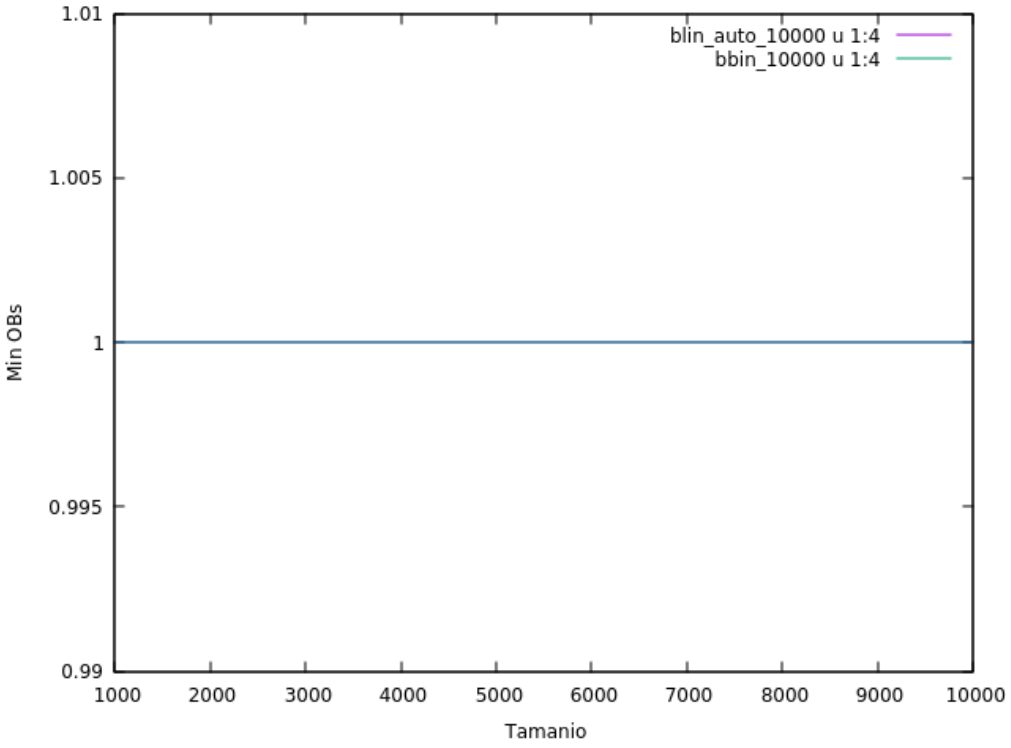
-n_veces = 1



-n_veces = 100



-n_veces = 10000



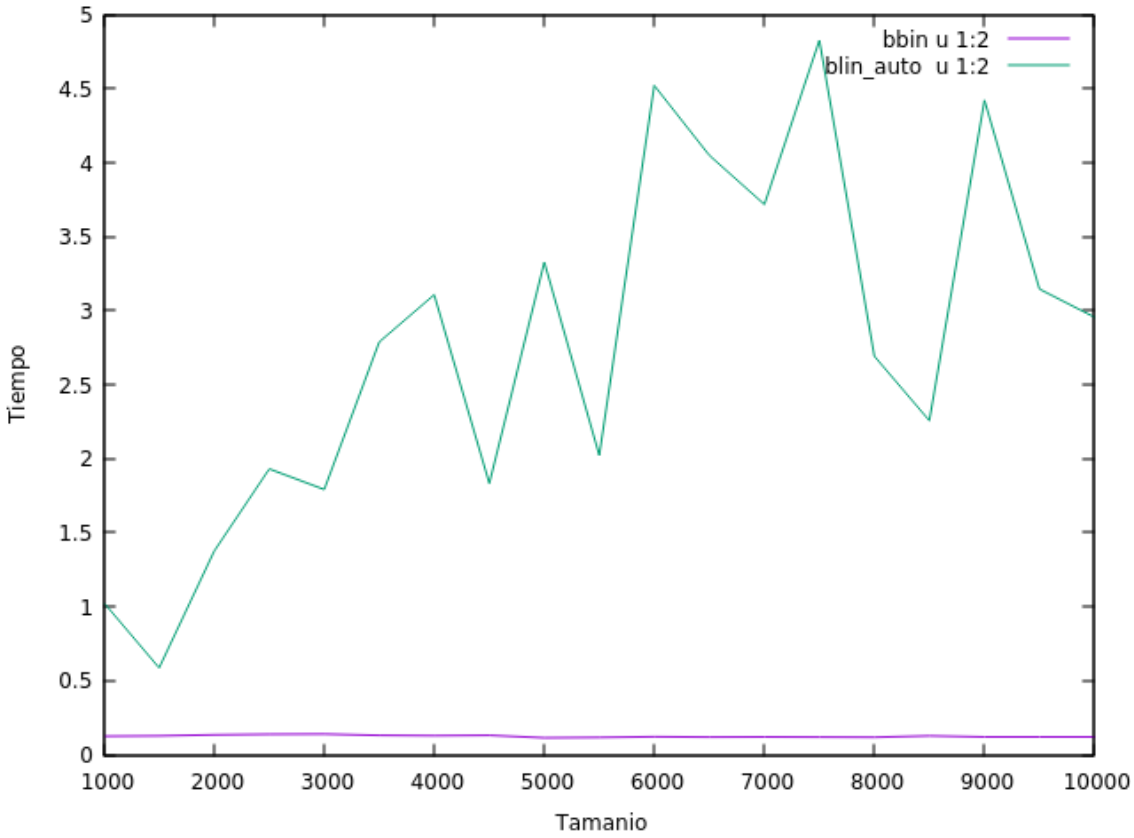
Al ver estas gráficas se puede observar que el número mínimo de OBs de bbin sobre diccionarios ordenados, aún con un generador de claves potencial, se mantiene bastante constante tanto si se busca cada clave 1 como 10000 veces.

En cambio, en el caso blin_auto, se observa una gran diferencia en el número mínimo de OBs cuando cada clave es buscada diferente número de veces. Esto es debido a la razón ya explicada anteriormente en esta memoria, es decir, a la peculiaridad de blin_auto, que cuando encuentra la clave buscada, intercambia esta con el elemento situado en la posición anterior.

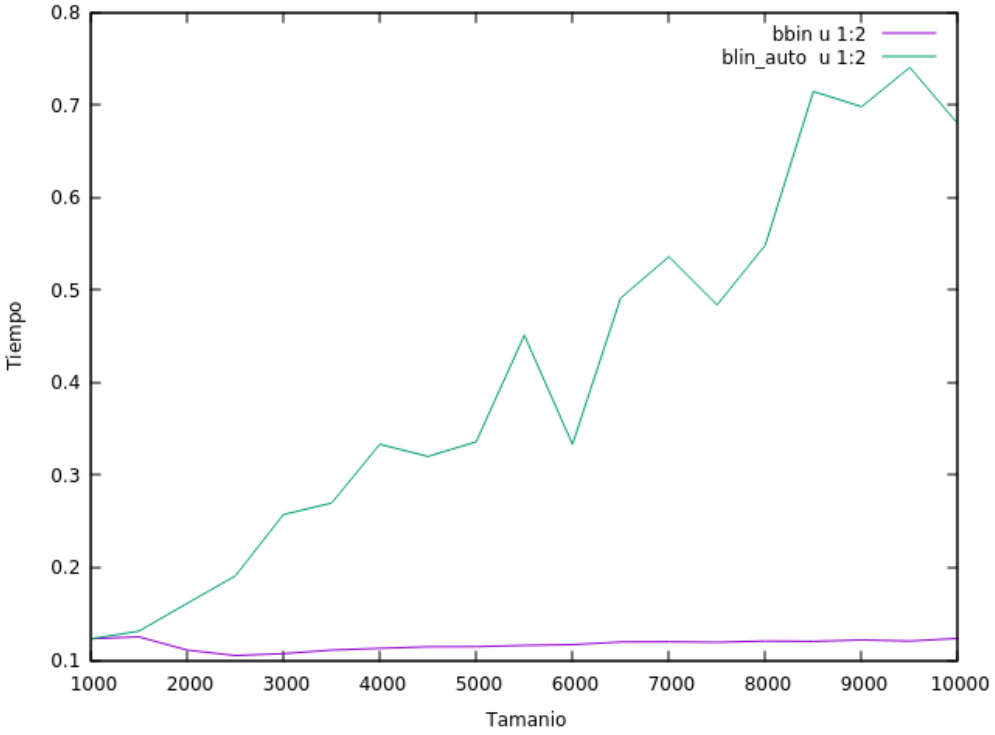
Se aprecia que ya con buscar cada clave 500 veces, es suficiente, para llegar al número mínimo de OBs de 1. Mientras que bbin, aún actuando sobre diccionarios ordenados, en el mismo caso, no llega a este mínimo de OBs. Mostrándose de nuevo el potencial de blin_auto.

Gráfica comparando el tiempo medio de reloj entre la búsqueda binaria y la búsqueda lineal auto organizada.

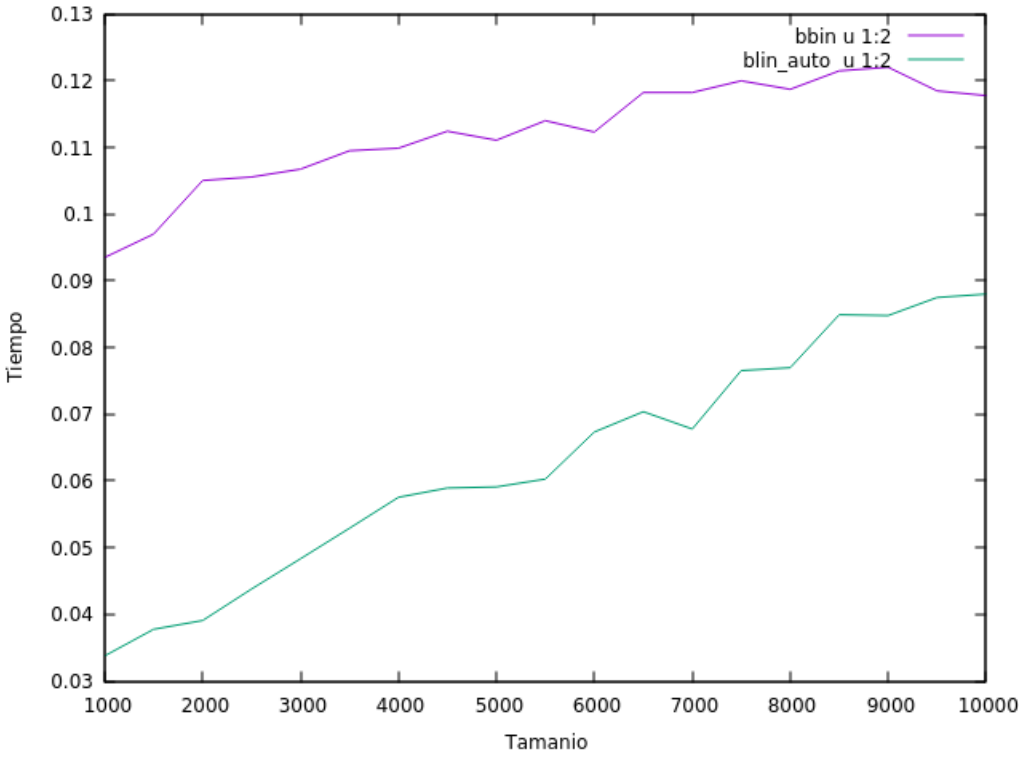
-n_veces = 1



-n_veces = 100



-n_veces = 10000



Una vez más se vuelve a comprobar la eficiencia de blin_auto cuando cada clave es buscada un gran número de veces, pero esta vez fijándonos en el tiempo medio.

Es apreciable que la velocidad de bbin para buscar claves un número pequeño de veces, supera con creces a la de blin_auto. Pero es cuando se busca cada clave un gran número de veces cuando se intercambian los papeles, y es blin_auto quién es mucho más rápido a la hora de encontrar claves.

Esto tiene mucho sentido, ya que cuando cada clave es buscada muchísimas veces, esta es colocada por el algoritmo al inicio de la tabla, así la. Próxima vez que se vuelva a buscar esta clave, será encontrada más rápido, y así sucesivamente.

5. Respuesta a las preguntas teóricas.

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

5.1 Pregunta 1

La operación básica, tanto de bbin, blin como blin_auto, es la misma, pues todos son del mismo tipo de búsqueda. La operación básica es la comparación entre claves.

5.2 Pregunta 2

Para la búsqueda lineal con éxito, sabemos que el caso peor ocurre cuando la clave se encuentra al final de la tabla, y el caso mejor es cuando se halla en la primera posición, luego los tiempos de ejecución son:

$W_{blin}(N) = O(N)$ y $B_{blin}(N) = O(1)$.

Como conocemos el número de operaciones necesarias para búsquedas binarias con éxito que el caso peor ocurre cuando la clave se encuentra en la última 'división' de la tabla luego los tiempos de ejecución son:

$W_{bbin}(N) = O(\log(N))$ y $B_{bbin}(N) = O(1)$.

5.3 Pregunta 3

Los elementos más frecuentes van desplazándose hacia el comienzo de la tabla, favoreciendo su búsqueda disminuyendo los tiempos y el número de operaciones básicas. Con la distribución no uniforme dada, esto provoca que tras suficientes búsquedas el tiempo medio para encontrar una clave disminuya enormemente, pues la gran mayoría de claves a buscar están en los comienzos de la tabla.

5.4 Pregunta 4

Una vez realizadas suficientes búsquedas como para que la lista de claves se quede en una situación estable, podemos asumir que el coste de búsqueda medio es de la forma O(1), pues la distribución potencial del generador de claves da preferencia a las que están al principio de la tabla, mientras que las que están al final rara vez se buscan.

5.5 Pregunta 5

Suponemos que tenemos un diccionario de claves ordenado. Sea T la lista de claves de tamaño N. Buscamos una clave k y sabemos que la lista está ordenada. Si suponemos que la clave k está dentro del diccionario, queremos ver que una búsqueda binaria siempre va a poder encontrarla.

La primera comparación con el elemento medio de la tabla nos indica en qué dirección se halla la clave k. Como siempre comparamos con el elemento medio de la subtabla en la que sabemos que se encuentra k, podemos ir precisando donde se encuentra k hasta encontrarlo. Esto es posible gracias a que el tamaño de la tabla es un entero finito, luego las divisiones por dos terminarán en algún punto; y la tabla está ordenada, luego si m es el elemento medio a comparar, si $T[m] < k \Rightarrow$ k está en la subtabla derecha, mientras que si $T[m] > k \Rightarrow$ k está en la subtabla izquierda.

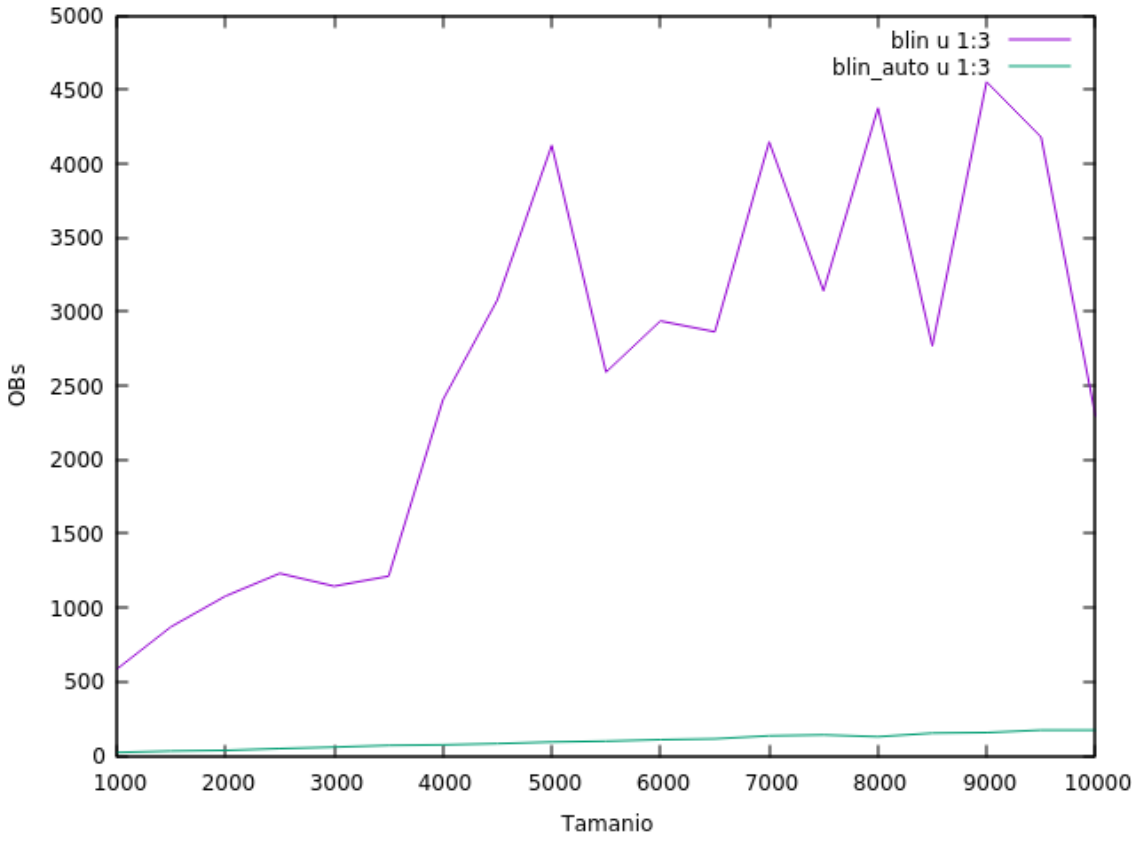
6. Conclusiones finales.

Esta práctica nos ha hecho darnos cuenta de una manera práctica de todos los contenidos explicados en el tema de diccionarios en clases teóricas.

Hemos sido capaces de implementar un diccionario que utilizaba tres diferentes algoritmos de búsqueda para buscar claves en tablas. Y nos hemos dado cuenta de lo útil y eficaz que resulta esto para buscar elementos tanto en tablas ordenadas como en no ordenadas.

También nos hemos percatado de qué algoritmo de búsqueda es más eficaz en ciertas situaciones, y cuáles lo son en otras. También hemos podido observar cómo una ligera modificación puede hacer a un algoritmo de búsqueda infinitamente más eficaz, como ocurre con la modificación realizada en blin para crear blin_auto.

En la siguientes gráficas se puede ver la eficacia de blin_auto frente a blin cuando cada clave es buscada varias veces (n_veces = 1000) y aplicados los algoritmos sobre diccionarios no ordenadas y con un generador de claves potencial.



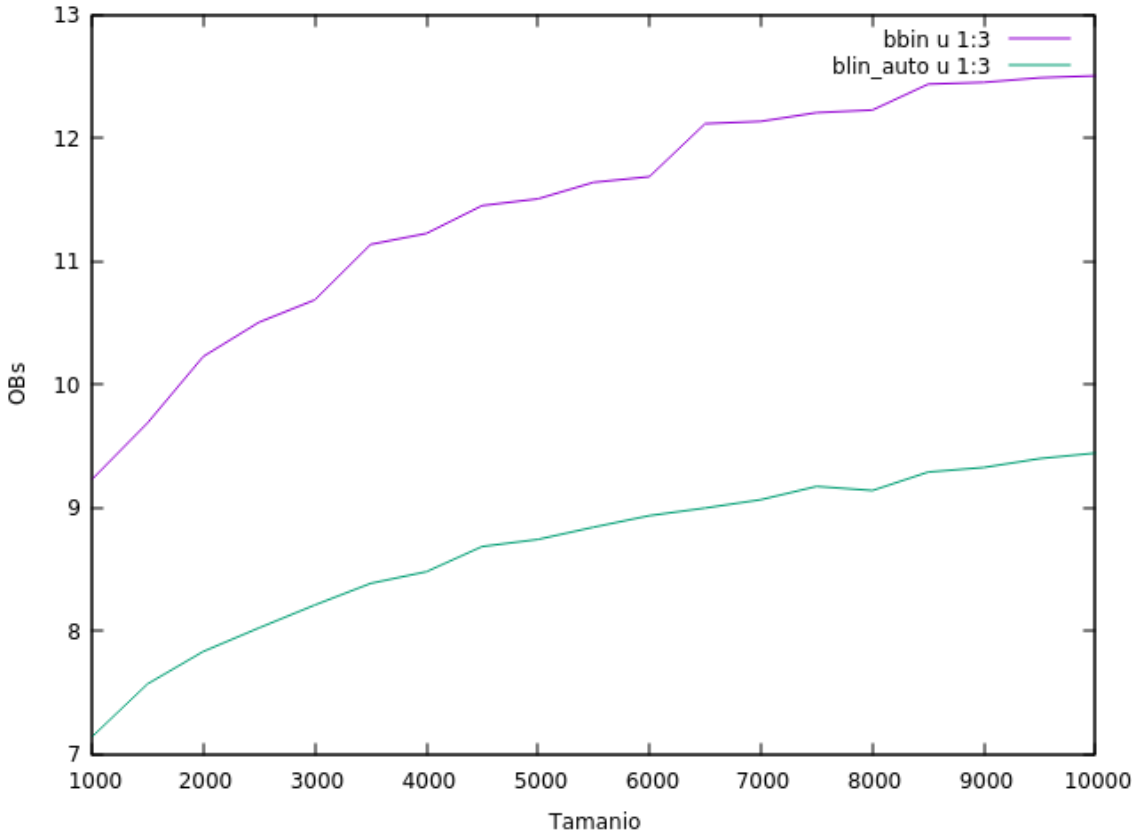
Una ligera modificación el código ha hecho que un algoritmo de búsqueda reduzca de esta manera en número medio de OBs.

Si, en cambio, cada clave fuese buscada una sola vez, las gráficas serían prácticamente idénticas.

Sin embargo también es evidente que el algoritmo óptimo de búsqueda cuando este es aplicado sobre diccionarios ordenados y cuando cada clave es buscada pocas veces, es sin ninguna duda bbin.

Pero si, en cambio, cada clave es buscada muchas veces, aunque el diccionario esté ordenado, blin_auto se convierte de nuevo en el más eficaz.

En las siguiente gráficas se puede apreciar una comparativa entre bbin y blin_auto ambos aplicados sobre diccionarios ordenados, con n_veces = 1000 y con un generador de claves potencial.



La gráfica cambiaría radicalmente si cada clave fuese buscada una sola vez, quedando la gráfica de bbin por debajo de la de blin_auto.

Después de todo este largo análisis, podemos concluir que el algoritmo de búsqueda más eficaz cuando una tabla está ordenada y cada clave es buscada un número pequeño de veces es bbin. Pero si, en cambio, las tablas no están ordenadas, o sí lo están pero el número de veces que se busca cada clave es elevado es blin_auto el algoritmo con mejor rendimiento de los tres.