Análisis de Algoritmos 2019/2020

Práctica 3

Claudia González Arteaga y Laura Iniesta Rodríguez 1261.

1. Introducción.

En esta tercera y última práctica de la asignatura de Análisis de Algoritmos vamos a implementar las funciones necesarias para el TAD diccionario. Entre ellas, se encuentra Busca Diccionario, para la cual implementaremos tres algoritmos de búsqueda(bbin, blin y blin_aut) que se encargarán de buscar una clave dentro de un diccionario. A continuación, implementaremos una serie de funciones para realizar medidas de rendimiento de las funciones de búsqueda desarrolladas anteriormente.

2. Objetivos

2.1 Apartado 1

El objetivo de este apartado es elaborar todas las funciones necesarias para la implementación del TAD diccionario: PDICC ini_diccionario (int tamanio,char orden); void libera_diccionario(PDICC pdicc); int inserta_diccionario(PDICC pdicc,int clave); int insercion_masiva_diccionario (PDICC pdicc,int *claves,int n_claves); int busca_diccionario(PDICC pdicc,int clave,int *ppos,pfunc_busqueda metodo).

Además, en este apartado se elaboran los algoritmos de búsqueda de Búsqueda lineal(sólo funciona cuando el diccionario esta ordenado), Búsqueda binaria, y Búsqueda binaria autoorganizada: int bbin(int *tabla,int P,int U,int clave,int *ppos); 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).

Todas las funciones devolverán el número de Operaciones Básicas que han empleado o ERR, en función de que su desarrollo fuera el deseado o no, exceptuando ini diccionario que devolverá NULL si ha ocurrido un error o bien el diccionario construido si todo se ha realizado correctamente.

2.2 Apartado 2

El objetivo de este apartado es implementar una serie de funciones para realizar medidas de rendimiento de las funciones de búsqueda desarrolladas en el apartado anterior. Para ello, elaboramos las funciones: short tiempo_medio_busqueda(pfunc_busqueda metodo, pfunc_generador_claves generador, char orden, int N, int n_veces, PTIEMPO ptiempo) y short genera_tiempos_busqueda(pfunc_busqueda metodo, pfunc_generador_claves generador, int orden, char* fichero, int num_min, int num_max, int incr, int n_veces).

3. Herramientas y metodología

El entorno que hemos usado para realizar esta práctica es Linux (Ubuntu), hemos programado en Atom y para la ejecución de los ejercicios hemos usado valgrind en la terminal de comandos para comprobar que no hay fallos de memoria, usamos Gnu plot desde la terminal para crear las gráficas de manera más eficiente.

3.1 Apartado 1

Para dar solución a este apartado que consistía en crear las funciones necesarias para un TAD diccionario y la funciones de búsqueda Bbin (Búsqueda Binaria) Blin (Búsqueda lineal) y Blin_auto (Búsqueda lineal autoorganizada) hemos usado la información dada en el enunciado de la práctica, junto con el pseudocódigo de inserción cuando el diccionario tiene que estar ordenado y la información sobre ellos que nos han dado en las clases de teoría y prácticas a lo largo de este cuatrimestre.

Ejecutamos el ejercicio 1 con todas las combinaciones posibles de las funciones creadas de búsqueda y ORDENADO o NO_ORDENADO, cuando se hace búsqueda de un número en una permutación ordenada podemos comprobar que es correcto cuando la posición que devuelve es la misma que su valor salvo en Blin_auto que es una posición menos debido al funcionamiento de esa función. Al usar Valgrind vemos que no hay fallos de memoria y que se libera toda la memoria que se ha reservado, comprobamos que al poner Bbin, NO_ORDENADO da error ya que esa función solo trabaja con permutaciones ordenadas.

3.2 Apartado 2

Para dar solución a este apartado hemos seguido los pasos que se indican en el enunciado de la práctica y nos hemos ayudado con las funciones que creamos en la práctica 1 ya que tienen el mismo funcionamiento.

Hemos implementado la función genera tiempos de búsqueda de forma que llama a la función encargada de buscar las claves y calcular las obs y el tiempo de ejecución y a la función encargada de imprimir en un fichero toda esa información a la vez luego, se generan los datos para un tamaño y se imprimen de seguido.

Hemos ejecutado el ejercicio 2 con valgrind comprobando que no hay fallos de memoria y que liberamos toda la memoria reservada para la realización de la práctica al igual que en el apartado anterior lo hemos hecho con todas las funciones de búsqueda ordenadas o no ordenadas.

Ejecutamos el ejercicio 2 repetidas veces variando los valores, poniendo números negativos cada vez en una entrada y poniendo un fichero NULL para comprobar el correcto funcionamiento de la función, también introducimos un número mínimo más alto que el máximo.

4. Código fuente

4.1 Apartado 1

```
PDICC ini_diccionario (int tamanio, char orden){
  PDICC dic = NULL;
  if(tamanio<0 || orden<0 || orden >1) return NULL;
  dic =(PDICC)malloc(sizeof(DICC));
  if(!dic) return NULL;
  dic->tamanio = tamanio;
  dic->n_datos = 0;
  dic->orden = orden;
  dic->tabla = (int*)malloc(sizeof(int)*tamanio);
  if(!(dic->tabla)){
          free(dic);
          return NULL;  }
  return dic;
}
void libera_diccionario(PDICC pdicc){
  if(!pdicc){
          return;  }
  pdicc->n_datos = 0;
    free(pdicc->tabla);
  free(pdicc);
}
```

```
int inserta_diccionario(PDICC pdicc, int clave){
  int j= 0, ob=1;
    if(pdicc == NULL) return ERR;
          pdicc->tabla[pdicc->n_datos] = clave;
  if(pdicc->orden == ORDENADO){
          j=pdicc->n_datos-1;
          while (j >=0 && pdicc->tabla[j]>clave){
                  ob++;
                  pdicc->tabla[j+1]= pdicc->tabla[j];
                  j--; }
          pdicc->tabla[j+1]=clave;
          ob++;  }
  pdicc->n_datos++;
  return ob;
}
int insercion_masiva_diccionario (PDICC pdicc,int *claves, int n_claves){
  int i=0, ob,flag;
    if(!claves || n_claves <0 || pdicc== NULL) return ERR;
          for(i=0;i<n_claves;i++){
          flag = inserta_diccionario(pdicc,claves[i]);
          if(flag == ERR) return ERR;
          ob+=flag; }
          return ob;
 }
```

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

  int p=0, u=0, flag = 0, ob=0;

  if(pdicc == NULL|| !ppos || !metodo) return ERR;

  u = (pdicc->n_datos)-1;

  if(pdicc->orden == NO_ORDENADO && metodo == bbin) return ERR;

  flag = metodo(pdicc->tabla,p,u,clave,ppos);

  if (flag == ERR) return ERR;

  ob = flag;

  return ob;

}
```

```
/* Funciones de busqueda del TAD Diccionario */
int bbin(int *tabla,int P,int U,int clave,int *ppos){
  int ob =0, m;
        if(!tabla || P<0 || U<0 || U<P || !ppos) return ERR;
        while(P<=U){
        m=(P+U)/2;
        if(tabla[m] == clave){
        ob++;
        (*ppos) = m;
        return ob;
        }else if(tabla[m] < clave){
        ob++;
        P=m+1;
        }else{
        ob++;
        U=m-1;
        }
        }
  *ppos = NO_ENCONTRADO;
  return ob;
}
```

```
int blin(int *tabla,int P,int U,int clave,int *ppos){
  int ob =0, i, j;
  if(!tabla || P<0 || U<0 || U<P || !ppos) return ERR;
  i=P;
  j=U;
  while(i<=j){
          if(tabla[i] == clave){
          ob++;
          (*ppos)=i;
          return ob;
          }else{
          ob++;
          i++; }
  }
  (*ppos)=NO_ENCONTRADO;
  return ob;
}
```

```
int blin_auto(int *tabla,int P,int U,int clave,int *ppos){
  int ob =0, aux = 0;
        if(!tabla || P<0 || U<0 || U<P || !ppos) return ERR;
        while(P<=U){
        if(tabla[P] == clave){
        ob++;
        if(P!=0){
        /*swap*/
        aux = tabla[P-1];
        tabla[P-1] = tabla[P];
        tabla[P] = aux;
        /*fin swap*/
        (*ppos)=P-1;
        return ob;
        }else{
        (*ppos)=P;
        return ob;
        }
        }else{
        ob++;
        P++;
        }
        }
        (*ppos)=NO_ENCONTRADO;
  return ob;
}
```

4.2 Apartado 2

```
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 pt = NULL;

  int i = 0;

  int flag = OK;

  if(!metodo|| !generador || orden< 0 || orden >1 || !fichero || num_min>num_max ||
num_max < 0 || num_min <0 || incr <0 || n_veces <0){

          return ERR;

  }

  pt = (PTIEMPO)malloc(sizeof(TIEMPO));

   if(!pt) return ERR;

   pt->N = 0;

  pt->n_elems = 0;

  pt->tiempo = 0;

  pt->medio_ob = 0;

  pt->min_ob = 0;

  pt->max_ob = 0;

   for(i = num_min; i <= num_max; i += incr){

          flag = tiempo_medio_busqueda(metodo, generador,orden, i,n_veces, pt);

          if(flag == ERR) return ERR;

          flag = guarda_tabla_tiempos(fichero, pt, 1);

          if(flag == ERR) return ERR;

    }

 free(pt);

 return flag; }
```

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

  PDICC pdicc=NULL;

  int *perm = NULL;

  int flag =0, ob =0, i;

  int pos, n;

  int* tabla = NULL;

  clock_t inicio, fin;

  if(!metodo || !generador || orden < 0 || orden > 1 || n_veces<=0 || !ptiempo) return ERR;

        n = N*n_veces;

        tabla = (int*)malloc(sizeof(int)*n);

        if(!tabla){

        return ERR;

        }

        perm = genera_perm(N);

        if(!perm) {

        free(tabla);

        return ERR;

        }

        pdicc = ini_diccionario(N, orden);

        if(!pdicc){

        free(tabla);

        free(perm);

        return ERR;     }
```

```
flag = insercion_masiva_diccionario(pdicc,perm,N);
if(flag == ERR){
libera_diccionario(pdicc);
free(perm);
free(tabla);
return ERR;
}
ob+=flag;
generador(tabla, n, N);
inicio = clock();
for(i=0; i< n; i++){
        ob = busca_diccionario(pdicc, tabla[i], &pos, metodo);
        ptiempo->medio_ob += ob;
        if(i == 0){
        ptiempo->min_ob = ob;
        ptiempo->max_ob = ob;
        }
        if(ptiempo->min_ob > ob){
        ptiempo->min_ob = ob;
        }
        if(ptiempo->max_ob < ob){
        ptiempo->max_ob = ob;
}
}
```

```
            fin = clock();
            ptiempo->tiempo = (fin - inicio);
            ptiempo->medio_ob /= n;
            ptiempo->n_elems = n;
            ptiempo->N = N;
    free(tabla);
    free(perm);
    libera_diccionario(pdicc);
  return OK;
}
```

5. Resultados, Gráficas

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

5.1 Apartado 1

Hacemos make y a continuación valgrind --leak-check=full ./ejercicio1 -tamanio 100 -clave 72

Para Blin con el diccionario ordenado:

Para Blin con el diccionario no ordenado:

Para Blin_auto con el diccionario ordenado:

```
laura@laura-UX331UA:~/Desktop/UAM/AALG/P3/codigo_p3(10-12)$ valgrind --leak-
k=full ./ejercicio1 -tamanio 100 -clave 72
==20493== Memcheck, a memory error detector
==20493== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==20493== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==20493== Command: ./ejercicio1 -tamanio 100 -clave 72
==20493==
Practica numero 3, apartado 1
Realizada por: Vuestros nombres
Grupo: Vuestro grupo
Clave 72 encontrada en la posicion 71 en 73 op. basicas
==20493==
==20493== HEAP SUMMARY:
==20493==     in use at exit: 0 bytes in 0 blocks
==20493==   total heap usage: 4 allocs, 4 frees, 1,848 bytes allocated
==20493==
==20493== All heap blocks were freed -- no leaks are possible
==20493==
==20493== For counts of detected and suppressed errors, rerun with: -v
==20493== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
laura@laura-UX331UA:~/Desktop/UAM/AALG/P3/codigo_p3(10-12)$
```

Para Blin_auto con el diccionario no ordenado:

```
laura@laura-UX331UA:~/Desktop/UAM/AALG/P3/codigo_p3(10-12)$ valgrind --leak-ch
k=full ./ejercicio1 -tamanio 100 -clave 72
==20340== Memcheck, a memory error detector
==20340== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==20340== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==20340== Command: ./ejercicio1 -tamanio 100 -clave 72
==20340==
Practica numero 3, apartado 1
Realizada por: Vuestros nombres
Grupo: Vuestro grupo
Clave 72 encontrada en la posicion 22 en 24 op. basicas
==20340==
==20340== HEAP SUMMARY:
==20340==     in use at exit: 0 bytes in 0 blocks
==20340==   total heap usage: 4 allocs, 4 frees, 1,848 bytes allocated
==20340==
==20340== All heap blocks were freed -- no leaks are possible
==20340==
==20340== For counts of detected and suppressed errors, rerun with: -v
==20340== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
laura@laura-UX331UA:~/Desktop/UAM/AALG/P3/codigo_p3(10-12)$
```

Para Bbin con el diccionario ordenado:

```
laura@laura-UX331UA:~/Desktop/UAM/AALG/P3/codigo_p3(10-12)$ valgrind --leak-chec
k=full ./ejercicio1 -tamanio 100 -clave 72
==20638== Memcheck, a memory error detector
==20638== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==20638== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==20638== Command: ./ejercicio1 -tamanio 100 -clave 72
==20638==
Practica numero 3, apartado 1
Realizada por: Vuestros nombres
Grupo: Vuestro grupo
Clave 72 encontrada en la posicion 72 en 6 op. basicas
==20638==
==20638== HEAP SUMMARY:
==20638==     in use at exit: 0 bytes in 0 blocks
==20638==   total heap usage: 4 allocs, 4 frees, 1,848 bytes allocated
==20638==
==20638== All heap blocks were freed -- no leaks are possible
==20638==
==20638== For counts of detected and suppressed errors, rerun with: -v
==20638== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
laura@laura-UX331UA:~/Desktop/UAM/AALG/P3/codigo_p3(10-12)$
```

Para Bbin con el diccionario no ordenado:

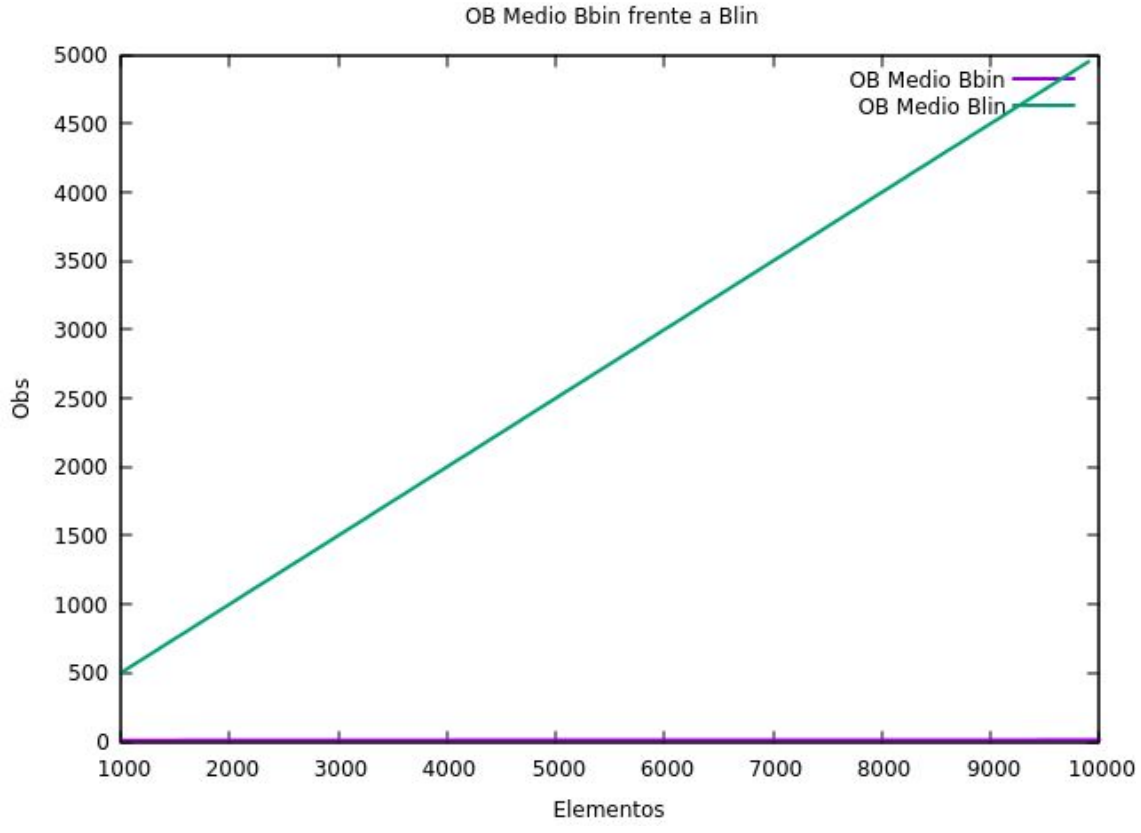
```
laura@laura-UX331UA:~/Desktop/UAM/AALG/P3/codigo_p3(10-12)$ valgrind --leak-che
k=full ./ejercicio1 -tamanio 100 -clave 72
==20615== Memcheck, a memory error detector
==20615== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==20615== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==20615== Command: ./ejercicio1 -tamanio 100 -clave 72
==20615==
Practica numero 3, apartado 1
Realizada por: Vuestros nombres
Grupo: Vuestro grupo

Bbin sólo funciona con diccionarios Ordenados.
Error al buscar la clave 72
==20615==
==20615== HEAP SUMMARY:
==20615==     in use at exit: 0 bytes in 0 blocks
==20615==   total heap usage: 4 allocs, 4 frees, 1,848 bytes allocated
==20615==
==20615== All heap blocks were freed -- no leaks are possible
==20615==
==20615== For counts of detected and suppressed errors, rerun with: -v
==20615== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
laura@laura-UX331UA:~/Desktop/UAM/AALG/P3/codigo_p3(10-12)$
```

5.2 Apartado 2

Ejecutamos el ejercicio dos con la siguiente linea de comando: valgrind --leak-check=full ./ejercicio2 -num_min 1000 -num_max 10000 -incr 100 -n_veces 1 -fichSalida ejercicio.log

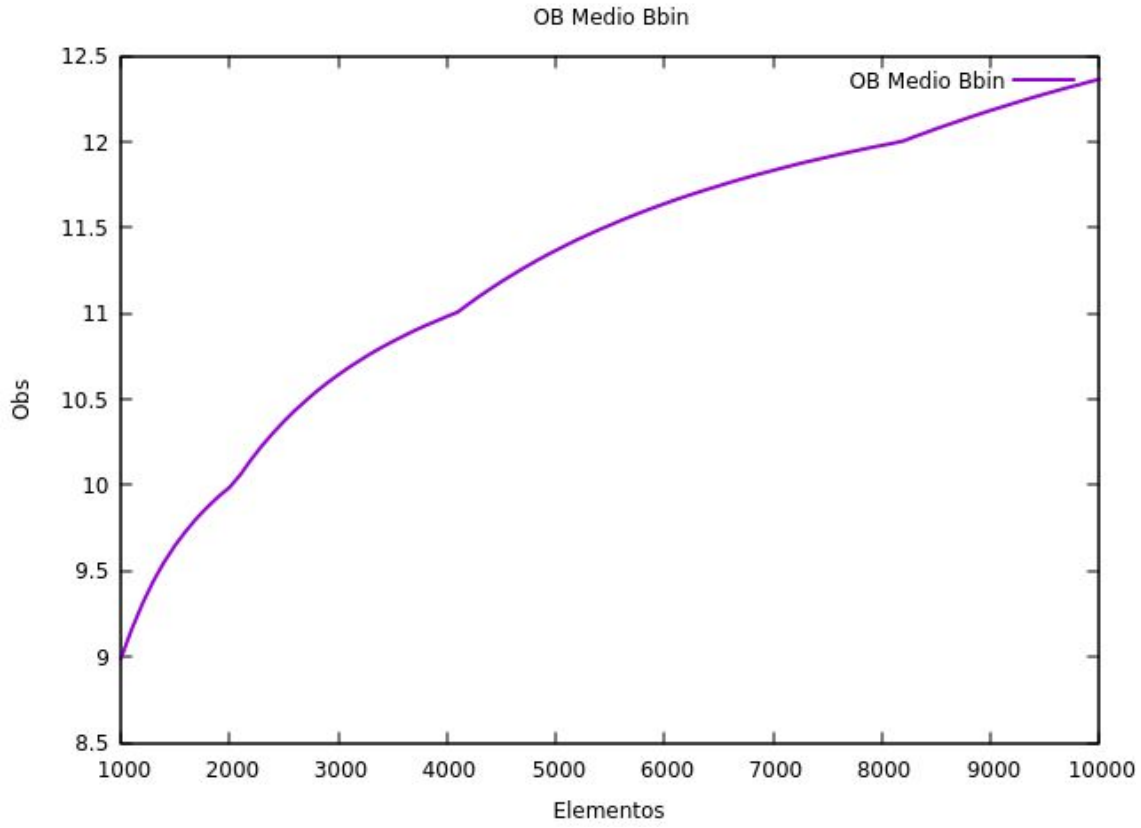
```
laura@laura-UX331UA:~/Desktop/UAM/AALG/P3/codigo_p3(12-12)$ valgrind --leak-chec
k=full ./ejercicio2 -num_min 1000 -num_max 10000 -incr 100  -n_veces 1 -fichSali
da ejercicio.log
==7317== Memcheck, a memory error detector
==7317== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==7317== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==7317== Command: ./ejercicio2 -num_min 1000 -num_max 10000 -incr 100 -n_veces 1
 -fichSalida ejercicio.log
==7317==
Practica numero 3, apartado 2
Realizada por: Claudia Gonzalez Arteaga y Laura Iniesta Rodriguez
Grupo: 1261
Salida correcta
==7317==
==7317== HEAP SUMMARY:
==7317==     in use at exit: 0 bytes in 0 blocks
==7317==   total heap usage: 542 allocs, 542 frees, 6,307,536 bytes allocated
==7317==
==7317== All heap blocks were freed -- no leaks are possible
==7317==
==7317== For counts of detected and suppressed errors, rerun with: -v
==7317== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
laura@laura-UX331UA:~/Desktop/UAM/AALG/P3/codigo_p3(12-12)$
```

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



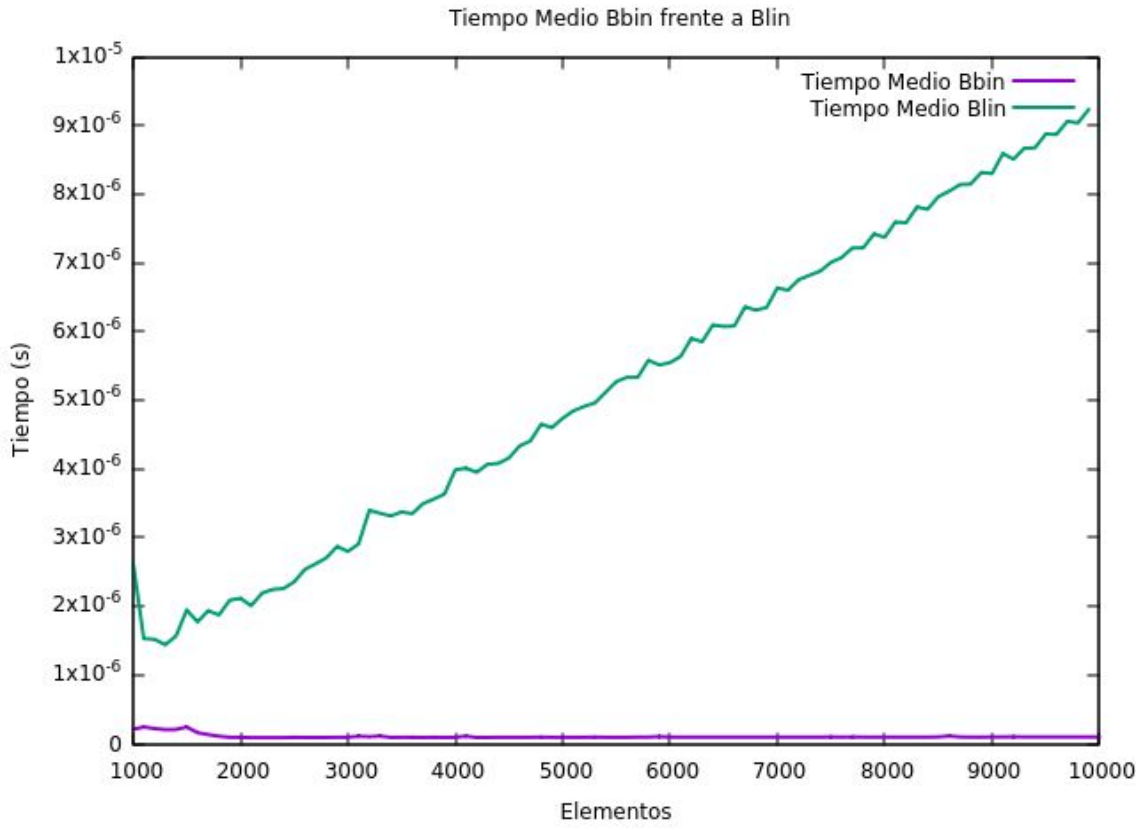
Como podemos observar, Bbin aparece en 0 todo el rato, esto es debido a que los valores medios de búsqueda binaria son muy pequeños en comparación con los de búsqueda lineal ya que esta comienza en 500 y búsqueda binaria en 9.

A continuación adjuntamos la gráfica con las obs de Bbin para que se pueda entender mejor esta gráfica.



Como podemos observar va desde 9 obs hasta aproximadamente 12.5, números insignificantes en comparación con 500 por lo que está línea que representa las obs de Bbin aparece en 0 en la gráfica anterior.

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



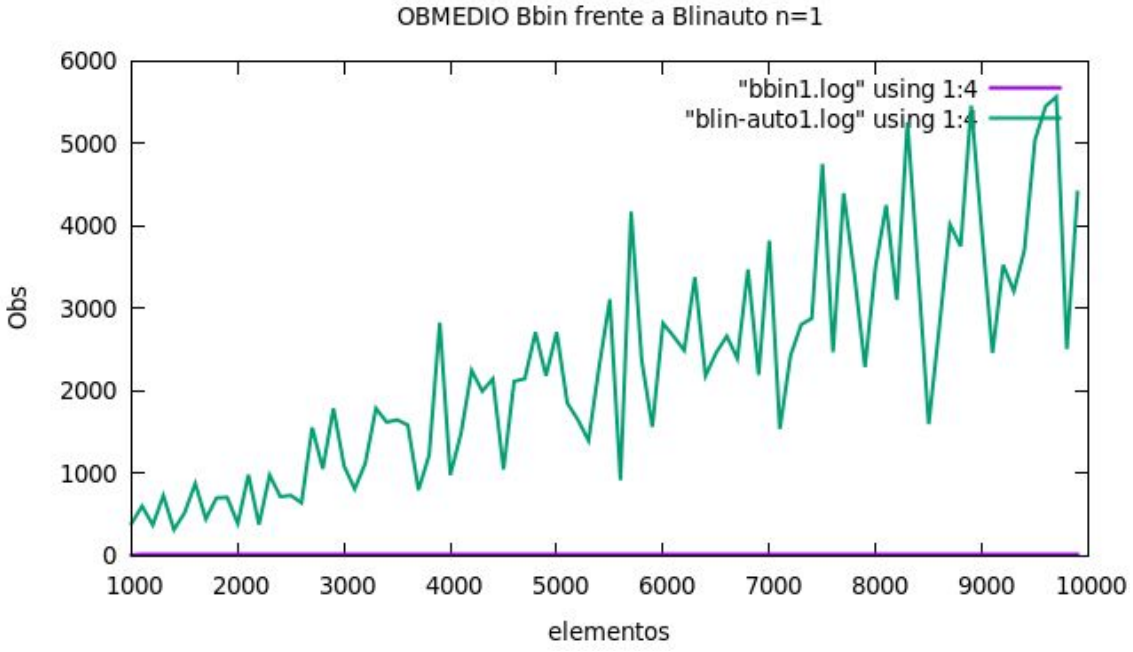
Como podemos observar pasa lo mismo que con la gráfica que compara las operaciones básicas promedio, el tiempo de ejecución de Bbin es tan pequeño en comparación con el de Blin que aparece en 0, aunque ambos tiempos son muy pequeños ya que no llegan ni a 1 msg.

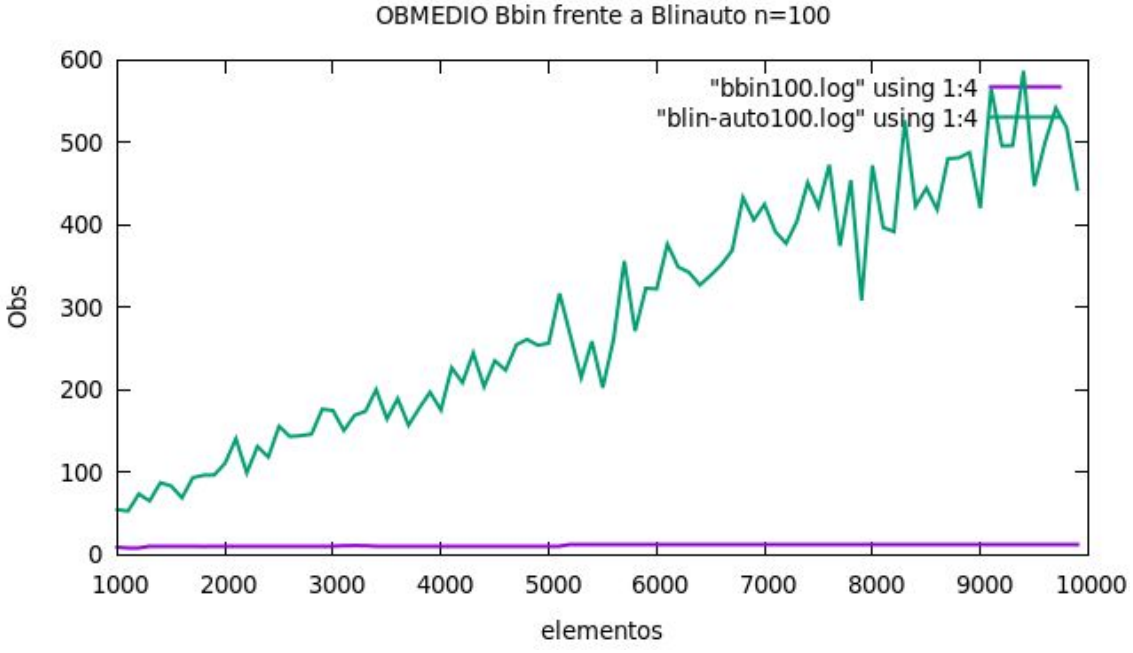
Como hemos visto en estas dos gráficas tanto Bbin como Blin son bastante eficientes aunque la búsqueda binaria es mucho mejor tanto en tiempo como en Obs que la búsqueda lineal.

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.

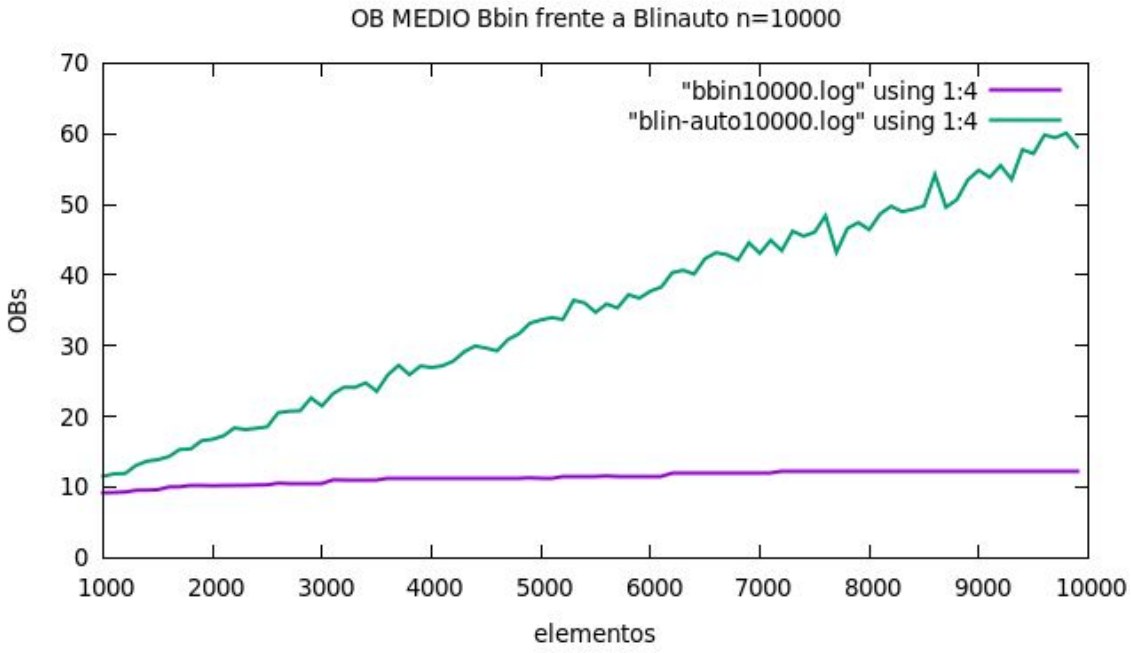
Verde: OB medio de Bbin.

Morado: OB medio de Blin_auto.





Verde: OB medio de Bbin. **Morado:** OB medio de Blin_auto.



Verde: OB medio de Bbin.

Morado: OB medio de Blin_auto.

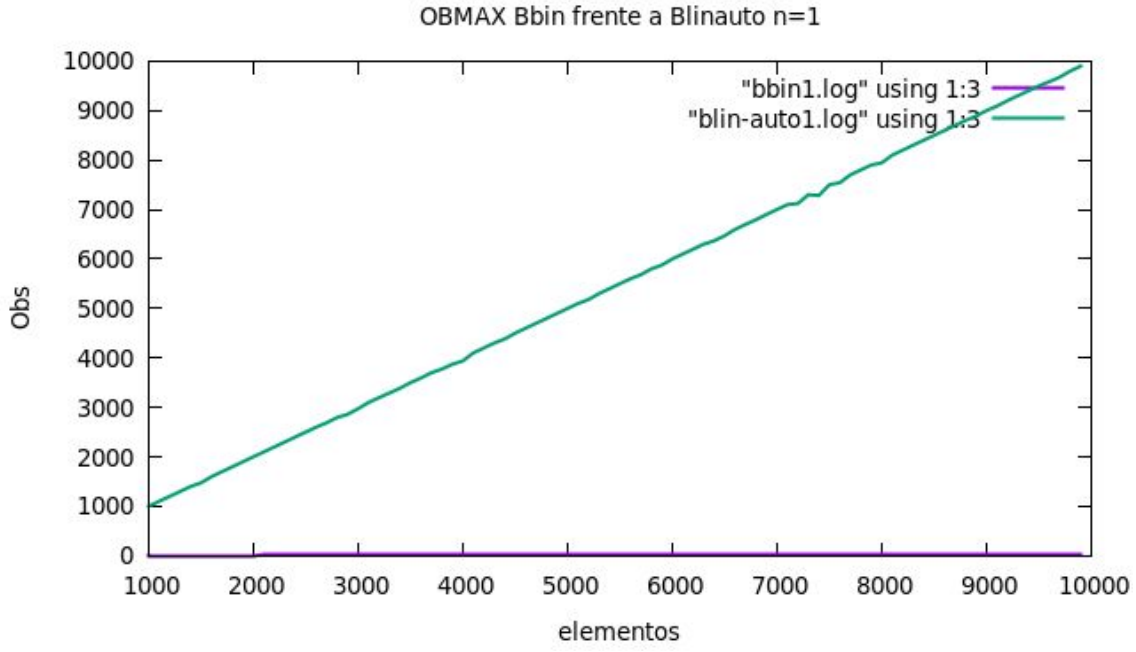
Como podemos observar en la gráfica, Bbin es mucho más eficiente que Blin auto ya que los valores son tan pequeños en comparación con los de Blin auto que aparecen en 0 al igual que en la primera gráfica.

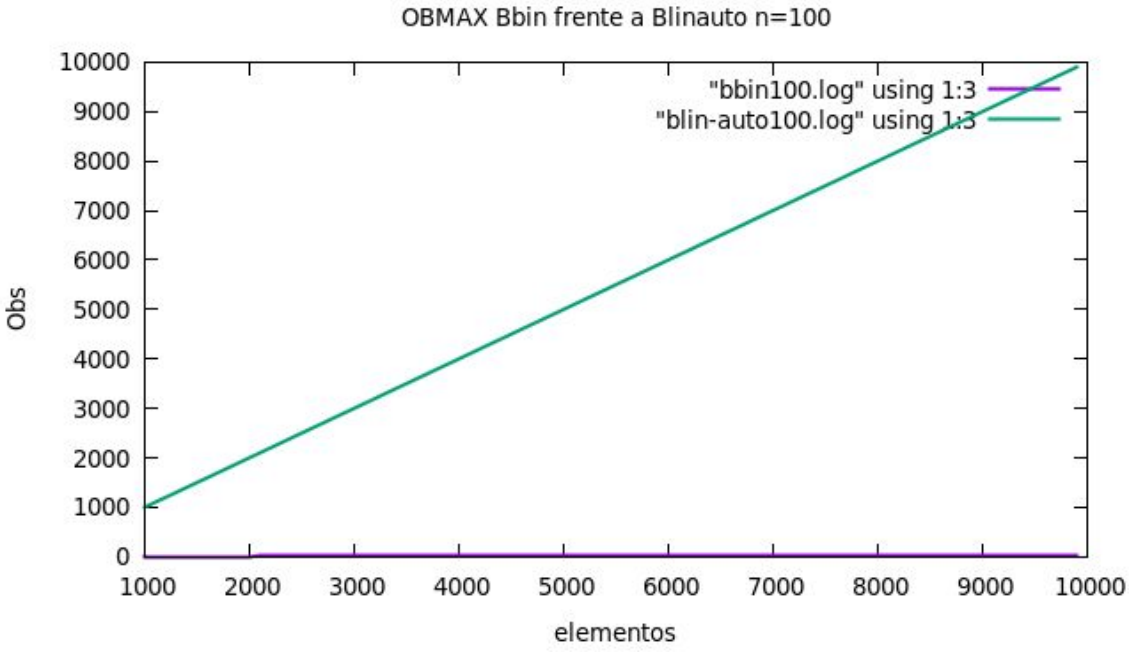
Debido al uso del generador de claves potencial a medida que aumenta nveces, las operaciones básicas son más uniformes, esto se debe a que el generador crea las claves con una probabilidad por lo que al realizar las búsquedas un mayor número de veces la búsqueda lineal auto organizada va colocando los números que salen con mayor probabilidad al principio del diccionario por lo que llega un momento que están ordenados por probabilidad, en cambio cuando las búsqueda se realiza pocas veces no da tiempo a que se ordenen por probabilidad, por eso al mirar las tres gráficas observamos como poco a poco es más uniforme.

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.

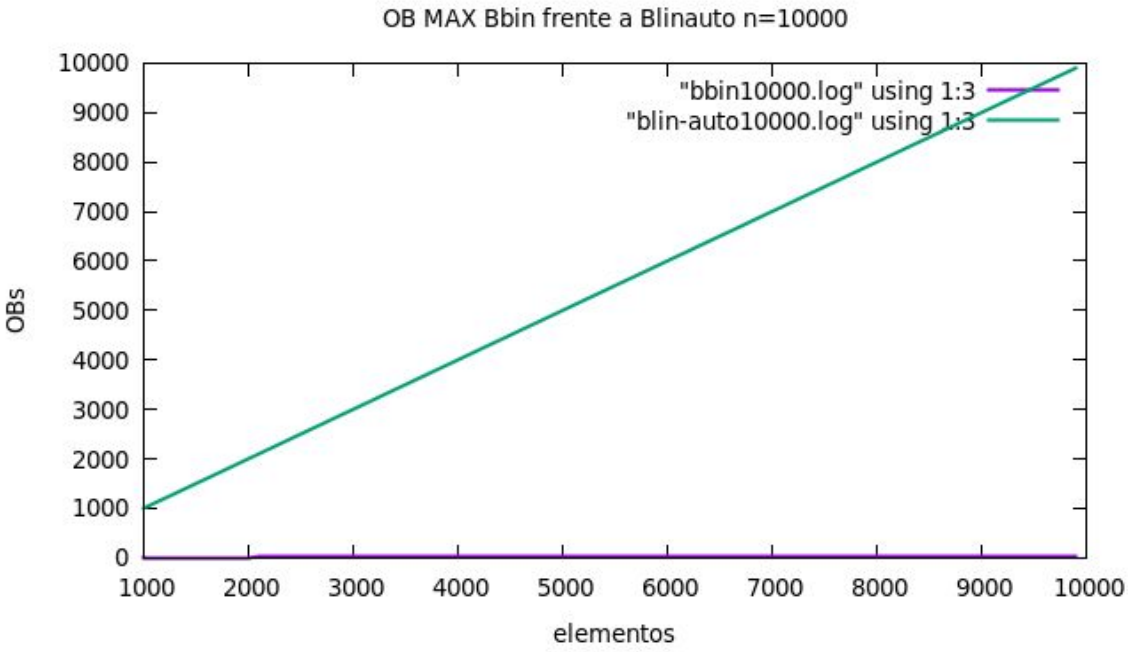
Verde: OB máximo de Bbin.

Morado: OB máximo de Blin_auto.





Verde: OB medio de Bbin. **Morado:** OB medio de Blin_auto.



Verde: OB medio de Bbin.

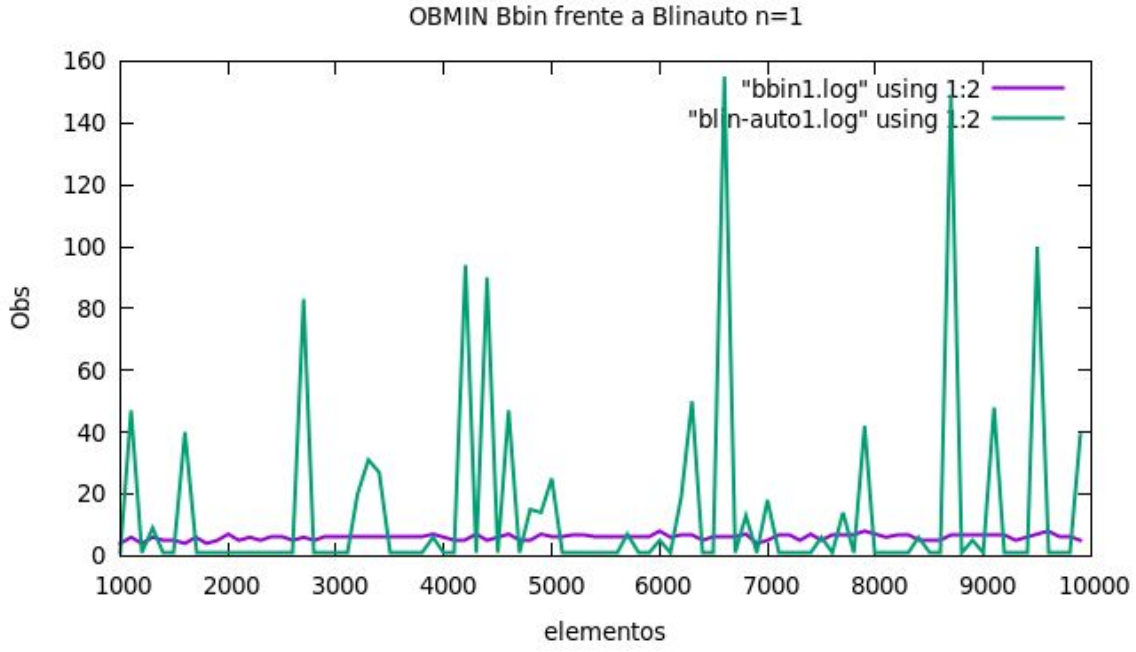
Morado: OB medio de Blin_auto.

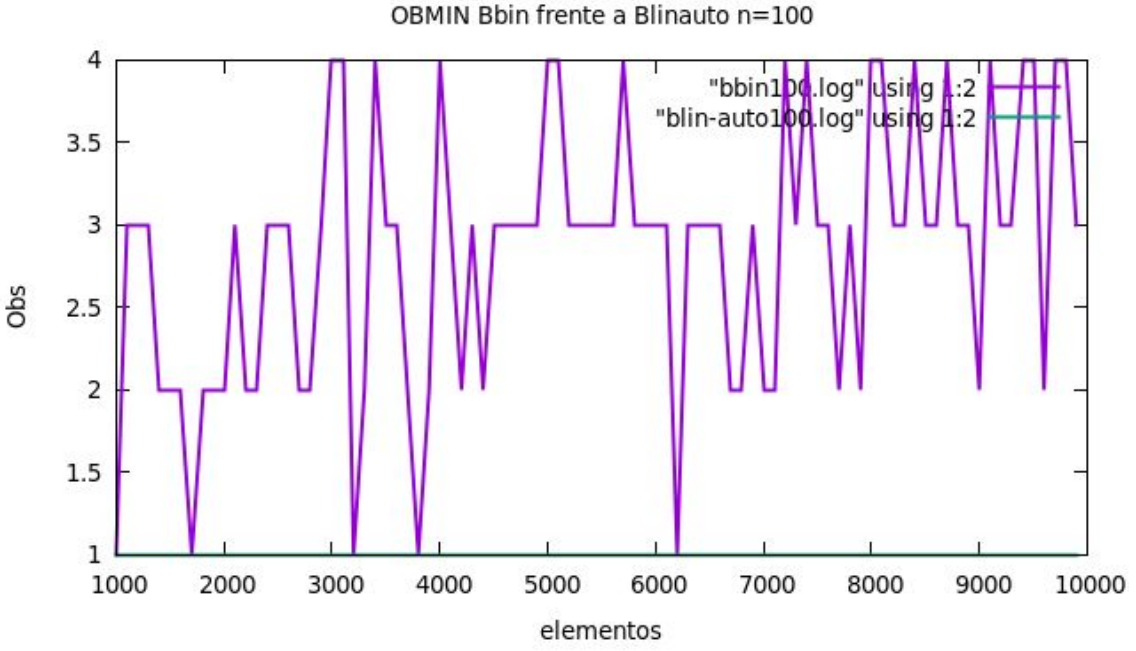
Como podemos observar búsqueda binaria es mucho más eficiente que búsqueda lineal autoorganizada al igual que en las gráficas anteriores, también podemos ver que las 3 gráficas son iguales sin importar el valor dado a nveces ya que el máximo es que tengan que recorrer toda la tabla porque el número está en la última posición, sin tener en cuenta el número de veces que se haga la búsqueda.

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.

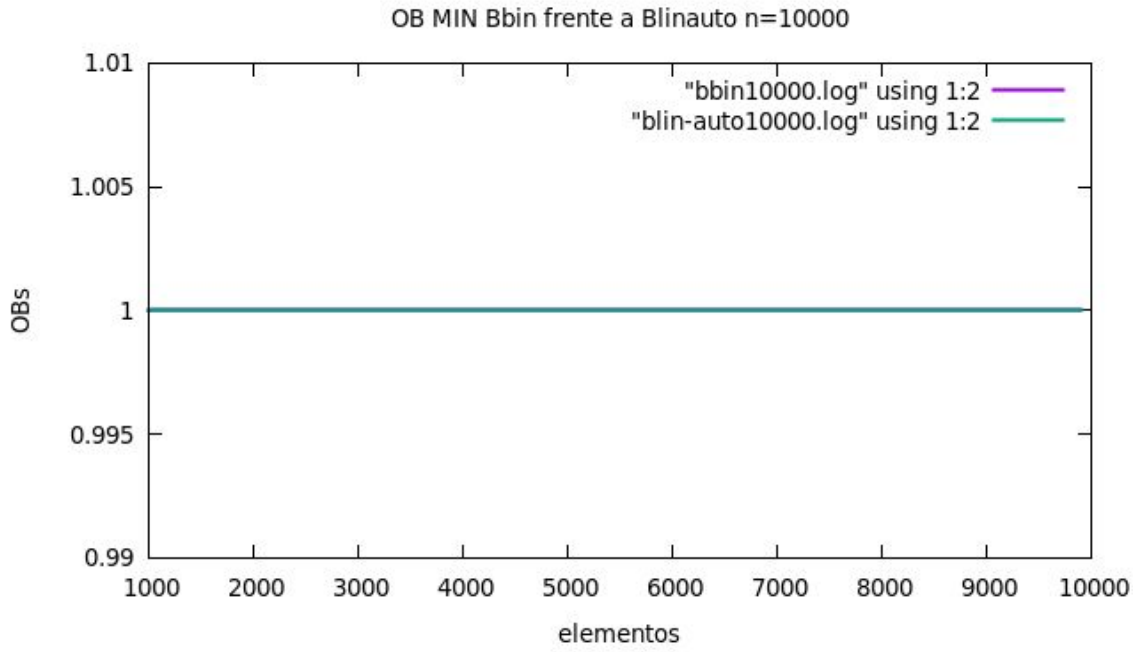
Verde: OB mínimo de Bbin.

Morado: OB mínimo de Blin_auto.





Verde: OB mínimo de Bbin. **Morado:** OB mínimo de Blin_auto.



Verde: OB mínimo de Bbin.

Morado: OB mínimo de Blin_auto.

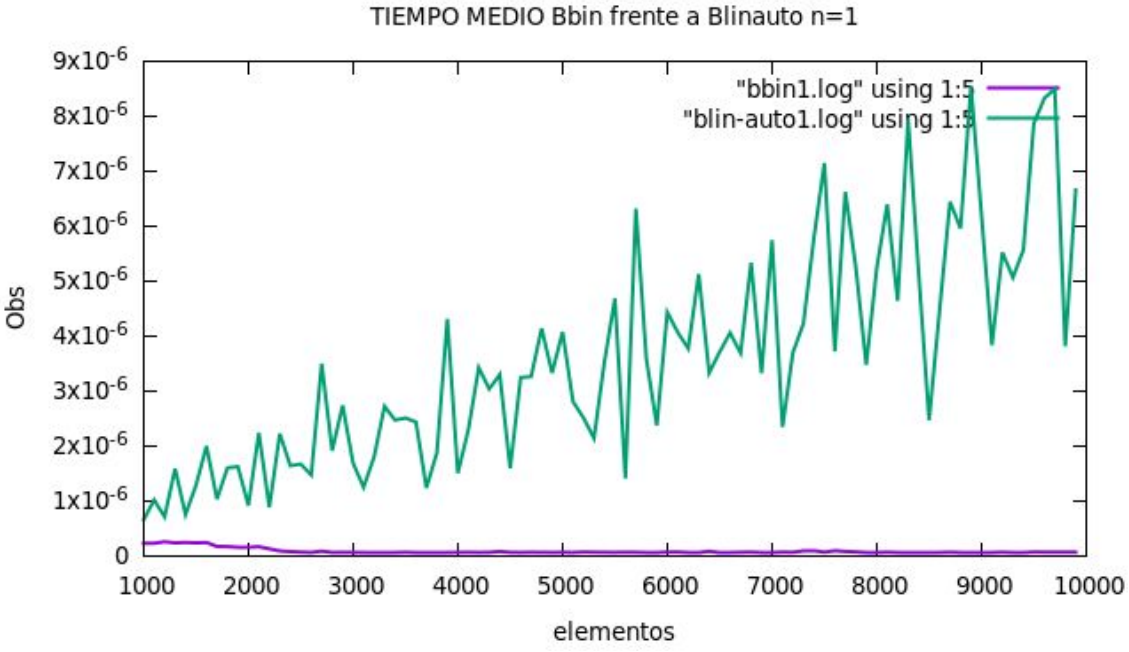
Como podemos observar en el caso de operaciones básicas mínimas BBin y Blin autoorganizada se parecen.

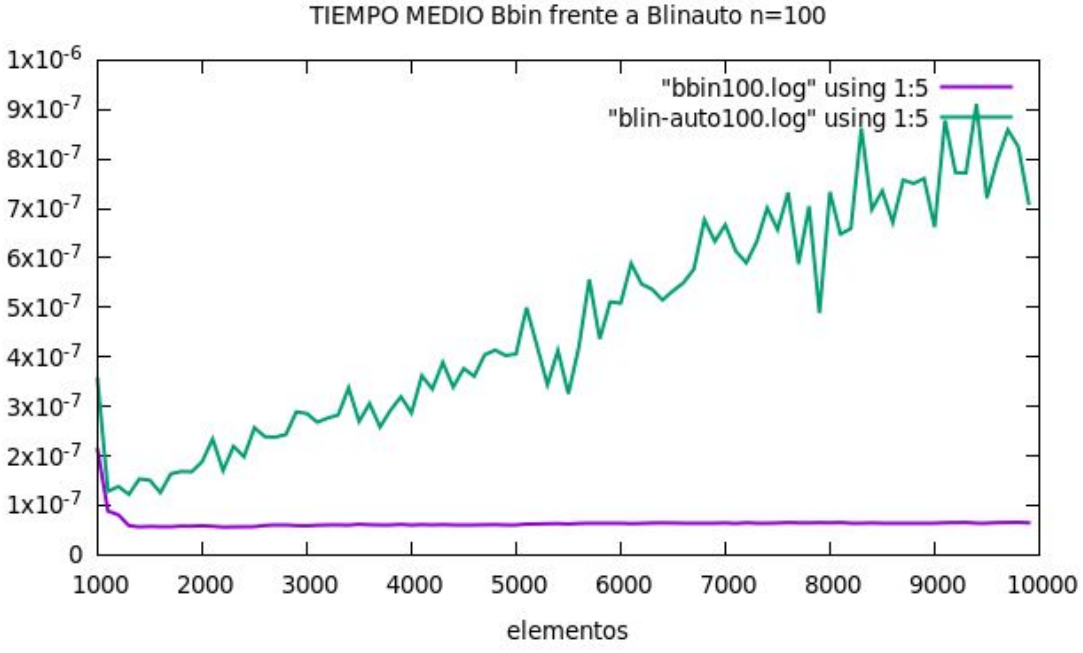
Debido al uso del generador de claves potencial a medida que aumenta nveces, las operaciones básicas son más uniformes, esto se debe a que el generador crea las claves con una probabilidad por lo que al realizar las búsquedas un mayor número de veces la búsqueda lineal auto organizada va colocando los números que salen con mayor probabilidad al principio del diccionario por lo que llega un momento que están ordenados por probabilidad, en cambio cuando las búsqueda se realiza pocas veces no da tiempo a que se ordenen por probabilidad, por eso al mirar las tres gráficas observamos como poco a poco es más uniforme, cómo estamos cogiendo las ob mínimas al ordenar la tabla llega un punto en el que sólo hace 1 ob ya que lo encuentra a la primera debido a la probabilidad con la que sale ese número.

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.

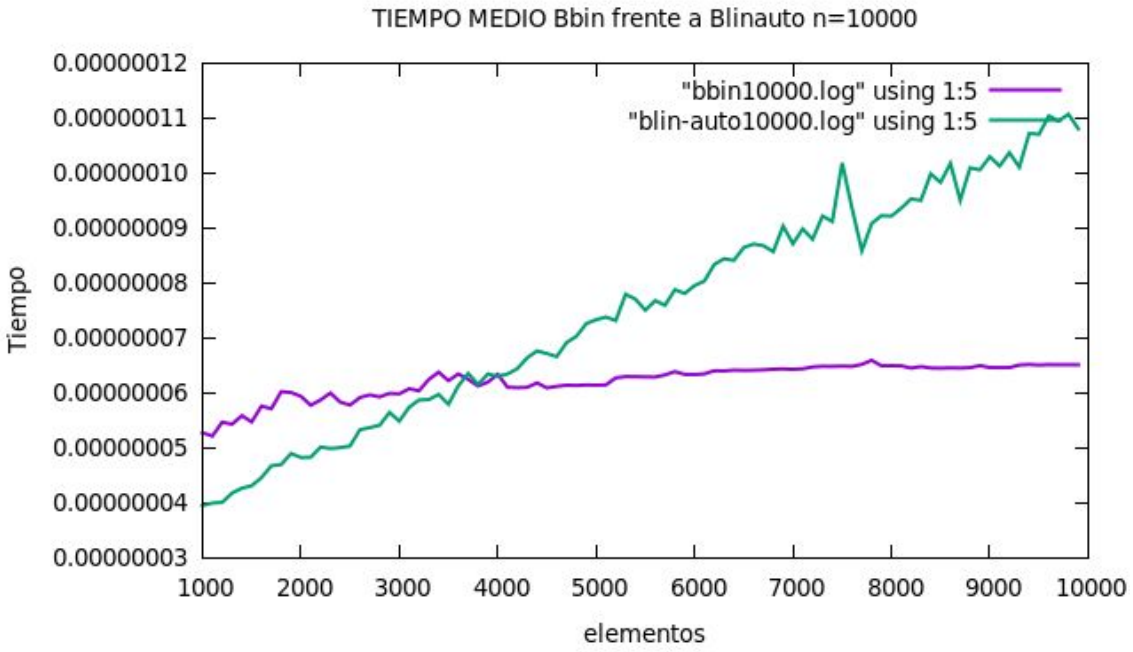
Verde: Tiempo medio de Bbin.

Morado: Tiempo medio de Blin_auto.





Verde: Tiempo medio de Bbin. **Morado:** Tiempo medio de Blin_auto.



Verde: Tiempo medio de Bbin.

Morado: Tiempo medio de Blin_auto.

Como podemos observar en la gráfica, Bbin es mucho más eficiente que Blin auto ya que los valores son tan pequeños en comparación con los de Blin auto que aparecen en 0 al igual que en la tercera gráfica (1ª gráfica de tiempos).

Debido al uso del generador de claves potencial a medida que aumenta nveces, el tiempo medio es más uniforme, esto se debe a que el generador crea las claves con una probabilidad por lo que al realizar las búsquedas un mayor número de veces la búsqueda lineal autoorganizada va colocando los números que salen con mayor probabilidad al principio del diccionario por lo que llega un momento que están ordenados por probabilidad, en cambio cuando las búsqueda se realiza pocas veces no da tiempo a que se ordenen por probabilidad, por eso al mirar las tres gráficas observamos como poco a poco es más uniforme.

5. Respuesta a las preguntas teóricas.

5.1 Pregunta 1

La operación básica de Búsqueda lineal, Búsqueda Binaria y Búsqueda binaria autoorganizada es la comparación de claves, es decir, cuando comparas si el elemento que estás buscando se encuentra en la posición indicada.

5.2 Pregunta 2

El caso peor y caso medio de Búsqueda binaria es $W=\lfloor \log N \rfloor = \log(n) + O(1)$

El caso peor de Búsqueda lineal es W=n y el caso medio es $Coste = n-1 \sum i=0 (i+1) 1/n = n+1/2$

5.3 Pregunta 3

Cuando se utiliza blin auto y la distribución no uniforme dada la posición de los elementos de la lista de claves varía según se van realizando más búsquedas ya que blin auto utiliza las propias claves a buscar para ir colocando las claves más comunes al inicio de la lista. Para que esto tenga efecto se deberán buscar muchas claves.

5.4 Pregunta 4

Teniendo en cuenta que blin autoorganizada realiza exactamente la misma búsqueda que el algoritmo de blin, que la única diferencia entre ambas es que blin auto coloca la clave en la posición anterior a la que la encuentra, mientras que el algoritmo blin la deja donde se encuentra y que esto no afecta a las OBs ya que en esa diferencia que tienen no se realiza comparación de claves, podemos concluir que el caso medio de Blin auto es el mismo que el de Blin, es decir, A=(n+1)/2.

5.5 Pregunta 5

El algoritmo de Búsqueda binaria “busca bien” porque realiza la búsqueda sobre una tabla ya ordenada y cuando no encuentra la clave en una mitad de la tabla, se va a la otra mitad, por lo que es un algoritmo muy eficiente ya que ahorra mucho tiempo.

6. Conclusiones finales.

En esta práctica hemos aprendido a implementar los algoritmos de búsqueda Bbin, Blin y Blin_auto asentando nuestros conocimientos teóricos sobre ellos y viendo su correcto funcionamiento, también hemos aprendido a usar el TAD diccionario pudiendo adquirir mayores conocimientos de programación y búsqueda de claves.

Al realizar las gráficas hemos podido comprobar que la búsqueda binaria es un algoritmos muy eficiente y que el algoritmo de búsqueda lineal auto organizada es más eficiente cuando las búsquedas se realizan varias veces ya que va moviendo al principio los elementos.