Análisis de Algoritmos 2017/2018

Práctica 3

Daniel Santo-Tomás y Lucia Rivas, 1201

Código	Gráficas	Memoria	Total

1. Introducción.

En esta práctica el objetivo es implementar el TAD Diccionario, comprendiendo su estructura y funcionamiento, para posteriormente desarrollar una serie de algoritmos de búsqueda y estudiar su rendimiento sobre diccionarios de distintos tamaños y con distintas claves.

2. Objetivos

2.1 Apartado 1

En este apartado implementamos el TAD Diccionario, desarrollando las funciones que aparecen en búsqueda.h. Además, implementamos los algoritmos de búsqueda binaria, búsqueda lineal y búsqueda lineal autoorganizada, probando su correcto funcionamiento en el ejercicio 1 que se nos suministra.

2.2 Apartado 2

En esta segunda parte ,desarrollamos funciones de medida de tiempo similares a las de la primera práctica ,con el objetivo de medir los tiempos de ejecución de los algoritmos de búsqueda sobre diccionarios de diversos tamaños,obteniendo además su número de operaciones básicas.

3. Herramientas y metodología

Para el desarrollo de esta práctica,hemos hecho uso de Atom como editor de texto en Linux,compilando con un Makefile y pasándole Valgrind a los programas para descartar problemas de memoria.

3.1 Apartado 1

Siguiendo la estructura diccionario dada en el enunciado,implementamos las funciones de diccionario.h. Ini_diccionario reserva memoria para el diccionario y para la tabla que contiene los datos,que será del tamaño pasado como argumento a la función. Libera_diccionario liberará la memoria de esta tabla y del diccionario.

Inserta_diccionario inserta una clave en un diccionario,situándola al final si es no ordenado y colocándola en su posición si es ordenado. Para colocarla,se inserta al final y se van desplazando las claves de la tabla del diccionario hacia la derecha hasta encontrar la posición correcta de la clave a insertar. Inserción_masiva inserta una serie de claves pasadas en un array ,llamando a inserta_diccionario tantas veces como claves haya(n_claves,pasado también como argumento).Ambas funciones devuelven el número de OBs necesarias para la inserción(la OB es la comparación).

Busca diccionario recibe como argumento un diccionario,un método de búsqueda,una clave y un puntero a int. Esta función llama al método sobre el diccionario,buscando la clave dada y pasando le también el puntero a int donde se guardará la posición de la clave. Devolverá lo que devuelva el método de búsqueda o ERR si hay algún error.

Los tres métodos de búsqueda implementados son búsqueda lineal(blin),búsqueda binaria(bbin) y búsqueda lineal autoorganizada (blin_auto).

Blin va comparando uno a uno y en orden los elementos de la tabla del diccionario con la clave a buscar,hasta encontrarla o llegar al fina de la tabla. Devolverá el número de OBs(la OB es la comparación) o NO_ENCONTRADO si la clave no está en el diccionario,y almacenará la posición de la clave(si la encuentra) en el puntero a int que se le pasa como argumento.

Bbin ,solo aplicable a diccionarios ordenados,empieza buscando en la posición media de la tabla((P+U)/2).Si la clave buscada está en esa posición,entonces la ha encontrado;en otro caso,si la clave es mayor,repite la búsqueda sobre la subtabla a la derecha del elemento medio(donde se encuentran las claves mayores).Si la clave es menor,repite la búsqueda nen la subtabla izquierda(donde se encuentran los elementos menores). Así sucesivamente hasta encontrar la clave o llegar a un punto donde no pueda buscar más. La función devuelve el número de OBs(la OB es la comparación) o NO_ENCONTRADO si la clave no está en el diccionario,y almacenará la posición de la clave(si la encuentra) en el puntero a int que se le pasa como argumento.

Blin_auto busca igual que blin,con la diferencia de que cuando encuentra una clave,la sitúa una posición a la izquierda(a menos que esté en la posición 0),de manera que la próxima vez que se busque esa clave,se precisen menos OBs para encontrarla. Devolverá el número de OBs(la OB es la comparación) o NO_ENCONTRADO si la clave no está en el diccionario,y almacenará la posición de la clave(si la encuentra) en el puntero a int que se le pasa como argumento.

Para probar el funcionamiento de bbin y blin,hemos ejecutado el ejercicio1 sobre un diccionario de tamaño 10 ordenado ,buscando la clave 1.

3.2 Apartado 2

En este apartado desarrollamos las funciones necesarias para el estudio de los tiempos y operaciones básicas de los algoritmos de búsqueda. Haciendo uso de la estructura tiempo ya utilizada en la práctica 1,desarrollamos dos funciones.

Tiempo_medio_búsqueda buscará ,con el método pasado como argumento , N claves ,n_veces cada una en un diccionario de tamaño N. Para ello,creamos el diccionario con el tamaño y el orden dado(se pasa como argumento),y con el generador de claves indicado ,guardamos en un array de tamaño N*n_veces un total de N*n_veces claves que van de 1 a N. A continuación,llamamos a busca_diccionario sucesivas veces sobre el diccionario ,buscando en cada iteración una clave del array creado. Medimos los tiempos de ejecución y almacenamos las OBs que devuelve el método,registrando también las máximas y mínimas. Para esto último,cada vez que se busca una clave,comparamos las OBs devueltas con el máximo y el mínimo que tengamos en ese momento almacenados,y actualizamos su valor si fuera necesario.

Para medir los tiempos,utilizamos la función gettimeofday .Se necesitan las librerías time.h y sys/time.h y la estructura timeval. Esta estructura consta de dos campos que registran tiempo,uno en segundos y otro en microsegundos. Para medir los tiempos,declaramos t_ini y t_fin,de tipo timeval,y registramos los datos de la hora,minuto,segundo etc antes de llamar a método mediante gettimeofday,guardándolo en t_ini. Después de la llamada a método,se registra de nuevo el tiempo,pero guardándolo en t_fin. Finalmente,con una conversión a double,se suman los campos

tv_sec(segundos) y tv_usec(microsegundos) de t_ini y t_fin,individualmente,y se resta el resultado de t_fin menos el de t_ini,sumándolo a un int llamado secs(inicializado a 0,va registrando el tiempo total de todas las llamadas a método).

Teniendo ya todos los datos,los almacenamos en la estructura tiempo de la siguiente forma:

ptiempo->N = N, tamaño del diccionario.

ptiempo->n_elems = N*n_veces, número de claves

ptiempo->max_ob = max,contador del máximo de OBs

ptiempo->min_ob = min,contador del mínimo de OBs

ptiempo->medio_ob = (med/(N*n_veces)),numero total de OBs entre número de claves,nos da el número medio de OBs.

ptiempo->tiempo = ((secs*1000000)/(N*n_veces)),tiempo de ejecución en segundos,multiplicado por 1000000 para pasarlo a microsegundos,y dividido entre el número de claves,nos da el tiempo promedio de ejecución en microsegundos. No lo dejamos en segundos porque entonces no se apreciaría en las gráficas.

Genera_tiempos_búsqueda es la segunda función,la cual ejecuta tiempo_medio_búsqueda sobre una serie de tablas de tamaño variable entre num_min y num_max(pasados como argumento),con un incremento de de valor también pasado como argumento. Almacenará los datos de cada ejecución en las sucesivas posiciones de una estructura ptiempo. Una vez almacenados todos los datos,se llama a la función guarda_tabla _tiempos,implementada en la práctica 1,que almacena la información de tamaño,tiempo de ejecución y OBs máximas,mínimas y promedias en un fichero pasado como argumento.

Para probar las funciones y generar los datos de cada método de búsqueda,hemos ejecutado el ejercicio2 varias veces. Primeramente,con el generador de claves uniforme,hemos estudiado bbin(en diccionarios ordenados) y blin(en diccionarios desordenados),con n_veces = 1 y con tamaños entre 1000 y 10000. Seguidamente,hemos estudiado bbin y blin_auto sobre diccionarios(ordenados en bbin y no ordenados en blin_auto) de tamaño también entre 1000 y 10000 ,pero con tres casos distintos de n_veces:1,100 y 10000. Los resultados y el estudio se ven reflejados en las gráficas de más abajo.

4. Código fuente

4.1 Apartado 1

```c
PDICC ini_diccionario (int tamanio, char orden)
{
  PDICC d;
  d = (PDICC)malloc(sizeof(DICC));
  if(!d) return NULL;
  d->tamanio = tamanio;
  d->orden = orden;
  d->tabla = (int*)malloc(sizeof(int)*tamanio);
  d->n_datos = 0;
  if(!d->tabla){
    free(d);
    return NULL;
  }
  return d;
}
```

```c
void libera_diccionario(PDICC pdicc)
{
  if(!pdicc) return;
  free(pdicc->tabla);
  free(pdicc);
  return;
}

int inserta_diccionario(PDICC pdicc, int clave)
{
  if(!pdicc) return ERR;
  pdicc->n_datos ++;
  if(pdicc->n_datos > pdicc->tamanio){
    pdicc->n_datos--;
    return ERR;
  }
  if(pdicc->orden == NO_ORDENADO){
      pdicc->tabla[pdicc->n_datos-1] = clave;
      return 0;
  }

  int cont = 0,i = 0,aux;
  pdicc->tabla[pdicc->n_datos-1] = clave;
  i = pdicc->n_datos-2;
  aux = clave;
  while(i >= 0 && pdicc->tabla[i] > aux ){
    cont++;
    pdicc->tabla[i+1] = pdicc->tabla[i];
    i--;
  }
  pdicc->tabla[i+1] = aux;

  return cont;
}
```

```c
int insercion_masiva_diccionario (PDICC pdicc,int *claves, int n_claves)
{
  if(!pdicc || !claves || n_claves < 1) return ERR;
  int i,cont = 0,contaux = 0;
  for(i = 0; i < n_claves; i++){
    contaux = inserta_diccionario(pdicc, claves[i]);
    if(contaux == ERR) return ERR;
    cont += contaux;
  }
  return cont;
}
```

```c
int busca_diccionario(PDICC pdicc, int clave, int *ppos, pfunc_busqueda metodo)
{
  if(!pdicc || !ppos) return ERR;
  int cont = 0;

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


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

```c
int blin(int *tabla, int P, int U, int clave, int *ppos)
{
  if(!tabla || !ppos) return ERR;
  int i, cont = 0;
  if(P > U){
    i = P;
    P = U;
    U = i;
  }
  for(i = 0; i <= U; i++){
    cont++;
    if(tabla[i] == clave) {
      *ppos = i;
      return cont;
    }
  }
  return NO_ENCONTRADO;
}
```

```c
int blin_auto(int *tabla,int P,int U,int clave,int *ppos){
  if(!tabla || !ppos) return ERR;
  int i, aux, cont = 0;
  if(P > U){
    i = P;
    P = U;
    U = i;
  }
  for(i = 0; i <= U; i++){
    cont++;
    if(tabla[i] == clave) {
      if(i == 0){
        *ppos = i;
        return cont;
      }
      aux = tabla[i];
      tabla[i] = tabla[i-1];
      tabla[i-1] = aux;
      *ppos = i - 1;
      return cont;
    }
  }
    return NO_ENCONTRADO;

}
```

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 || !generador) return ERR;

PDICC d;
int *perm,*claves;
int max = -1, min = -1, i, aux = 0, ppos;
double secs = 0, med = 0;
struct timeval t_ini, t_fin;

d = ini_diccionario(N,orden);
if(!d) return ERR;
perm = genera_perm(N);
if(!perm){
  free(d);
  return ERR;
}

if(insercion_masiva_diccionario (d, perm, N) == ERR){
  free(perm);
  free(d);
  return ERR;
}

claves = (int*)malloc(sizeof(int)*N*n_veces);
if(!claves){
  free(d);
  return ERR;
}

generador(claves, n_veces*N, N);

for(i = 0; i < N*n_veces; i++){
  gettimeofday(&t_ini, NULL);
  aux = busca_diccionario(d, claves[i], &ppos, metodo);

```

```
    gettimeofday(&t_fin, NULL);
    if(aux == ERR){
      free(claves);
      free(d);
      return ERR;
    }

    secs += (double)(t_fin.tv_sec + (double)t_fin.tv_usec/1000000) - (double)(t_ini.tv_sec + (double)t_ini.tv_usec/1000000);
    med += aux;
    if(min == -1 && max == -1) {
      min = aux;
      max = aux;
      continue;
    }

    if(aux < min) min = aux;
    if(aux > max) max = aux;

  }

  ptiempo->N = N;
  ptiempo->n_elems = N*n_veces;
  ptiempo->max_ob = max;
  ptiempo->min_ob = min;
  ptiempo->medio_ob = (med/(N*n_veces));
  ptiempo->tiempo = ((secs*1000000)/(N*n_veces));

  free(claves);
  libera_diccionario(d);
  free(perm);

  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 || !generador || !fichero || num_min <=0 || num_max <= num_min || incr <= 0 || n_veces <= 0 ) return ERR;

    int i;

    PTIEMPO tiempo = (PTIEMPO)malloc(sizeof(TIEMPO)*(((num_max - num_min)/incr) + 1));
    PTIEMPO aux;
    if(!tiempo) return ERR;

    aux = tiempo;

    for(i = num_min ; i <= num_max ; i += incr){
      printf("tamaño %d\n",i);
      if(tiempo_medio_busqueda( metodo, generador, orden, i, n_veces, aux)== ERR) return ERR;
      aux++;
    }

    if(guarda_tabla_tiempos(fichero, tiempo, (((num_max - num_min)/incr) + 1)) == ERR) return ERR;

    free(tiempo);

    return OK;
  }
```

5. Resultados, Gráficas

5.1 Apartado 1

Ejercicio 1 con búsqueda binaria:

```
danist@danist-Lenovo-E51-80:~/Documentos/UAM/PRACTICAS ANALISIS/ANAL/P3/codigo_p3$ ./ejercicio1 -tamanio 10 -clave 1
Practica numero 3, apartado 1
Realizada por: Lucía Rivas y Daniel Santo-Tomás
Grupo: 1201
Clave 1 encontrada en la posicion 0 en 3 op. basicas
```

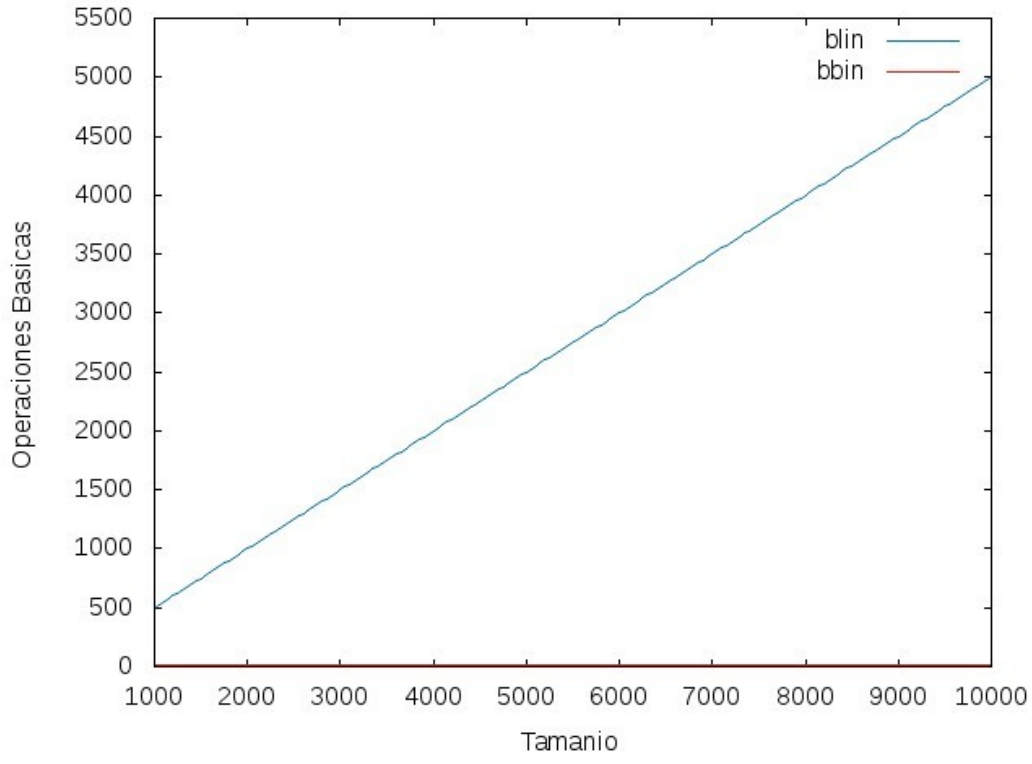
Ejercicio 1 con búsqueda lineal:

```
danist@danist-Lenovo-E51-80:~/Documentos/UAM/PRACTICAS ANALISIS/ANAL/P3/codigo_p3$ ./ejercicio1 -tamanio 10 -clave 1
Practica numero 3, apartado 1
Realizada por: Lucía Rivas y Daniel Santo-Tomás
Grupo: 1201
Clave 1 encontrada en la posicion 0 en 1 op. basicas
```

5.2 Apartado 2

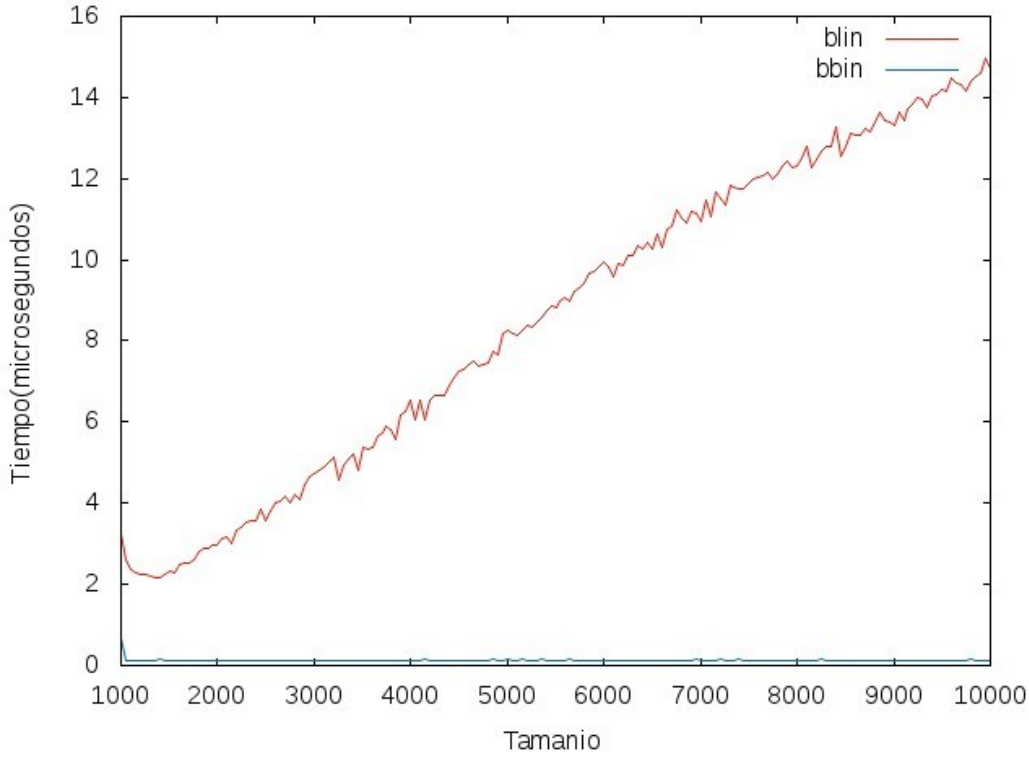
Las gráficas quedan tan picudas debido a que estudiamos una única permutación

Número promedio de OBs de la búsqueda lineal y la búsqueda binaria



Como se puede observar,y como pasará en la mayoría de las gráficas,la eficiencia de bbin es bastante mejor que la de blin,hasta el punto que a escala ,apenas es perceptible la gráfica de bbin,que roza el 0,aunque en realidad va desde las 9 OBs de promedio mínimo hasta un máximo de 12. Blin sin embargo crece linealmente, alcanzando en cada tamaño un número promedio de OBs de aproximadamente tamaño/2.

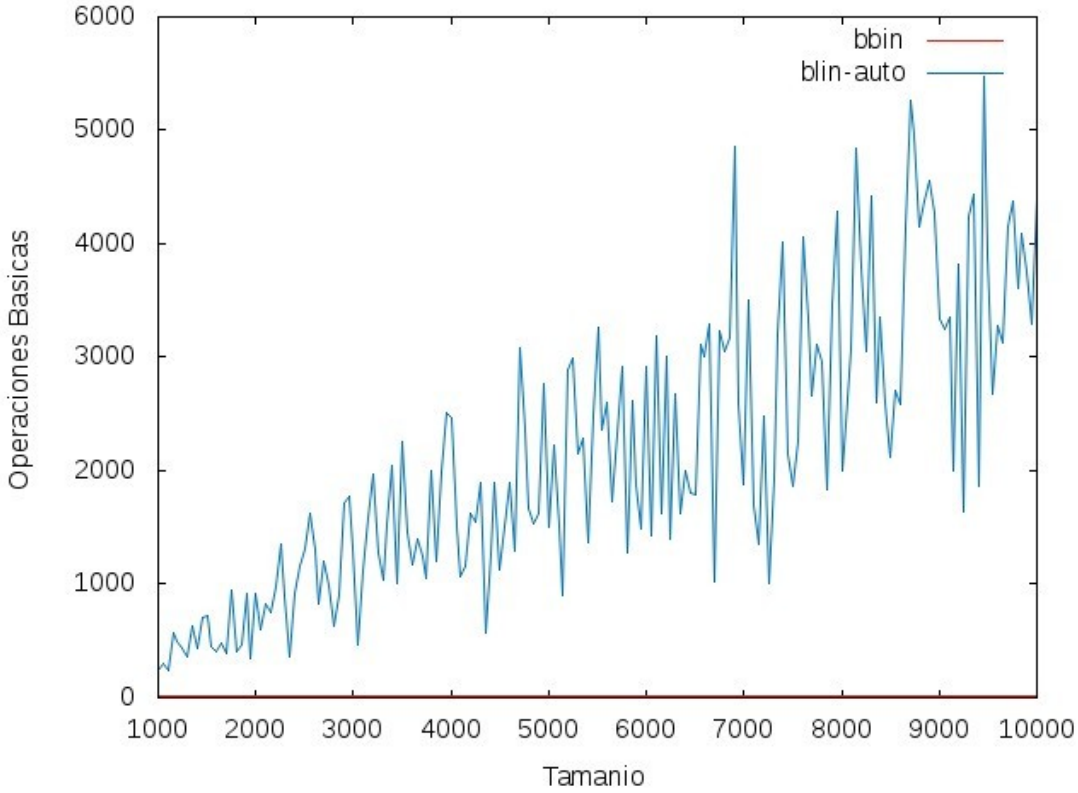
Tiempo promedio de reloj de la búsqueda lineal y la búsqueda binaria.



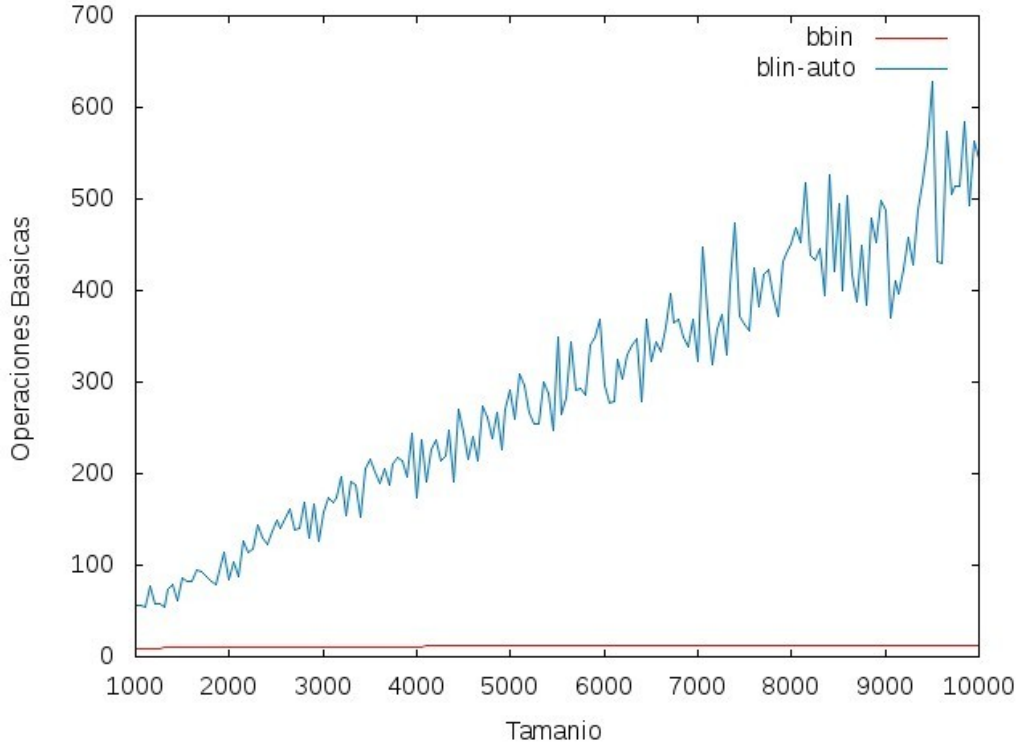
Al igual que con las Obs,al ser bbin más eficiente,apenas se ve en comparación con los tiempos de blin,cuyos tiempo crecen más rápido que los de bbin a medida que aumenta el tamaño,alcanzando un máximo de 14.96 microsegundos,mientras que bbin llega solo hasta los 0,13 microsegundos. La pequeña bajada de tiempo que se ve en ambas gráficas se debe a que son las primeras búsquedas que realiza el procesador nada más ejecutar el programa,por lo que tardan algo más.

Número promedio de OBs de la búsqueda binaria y la búsqueda lineal auto organizada

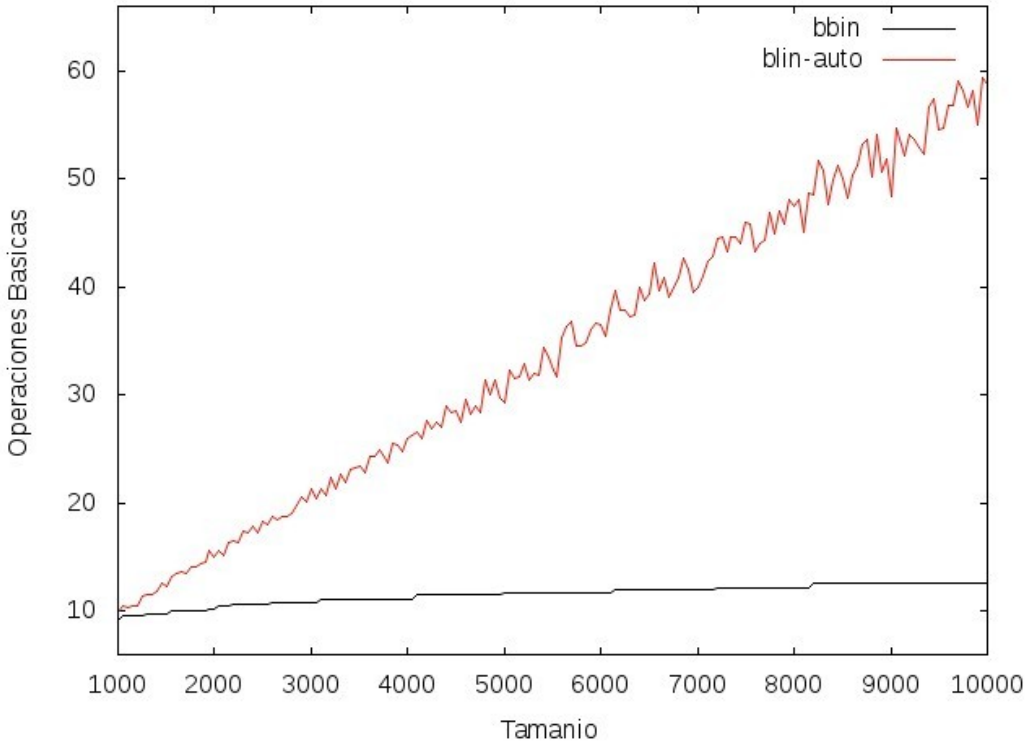
n_veces = 1



n_veces = 100



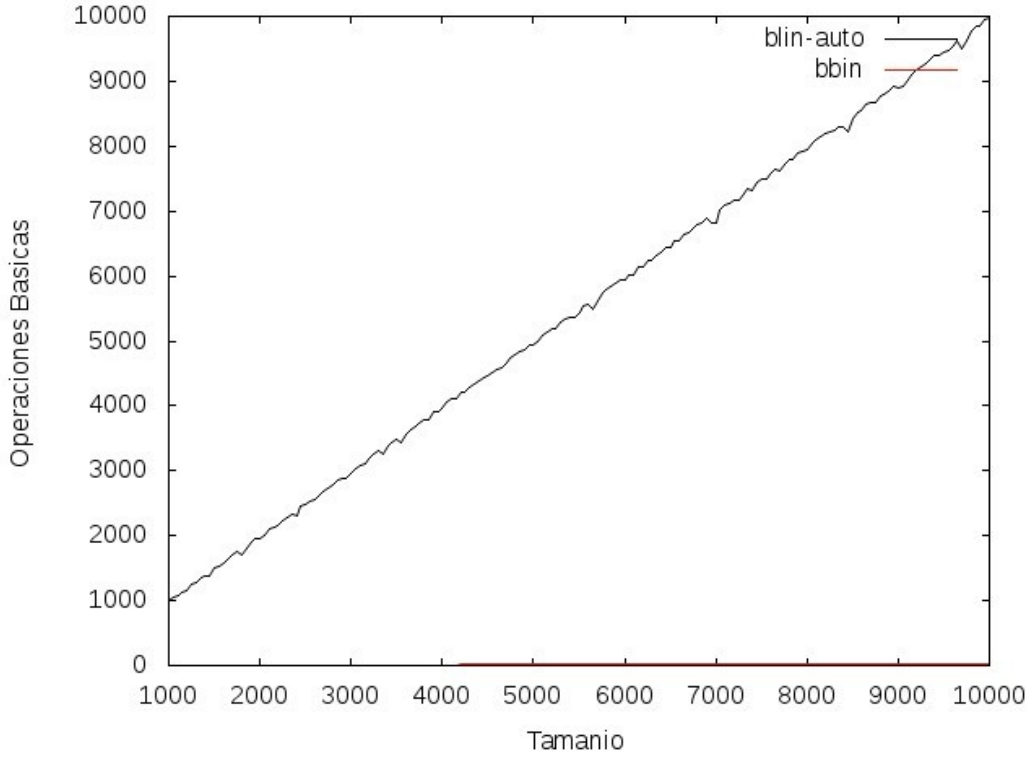
n_veces = 10000



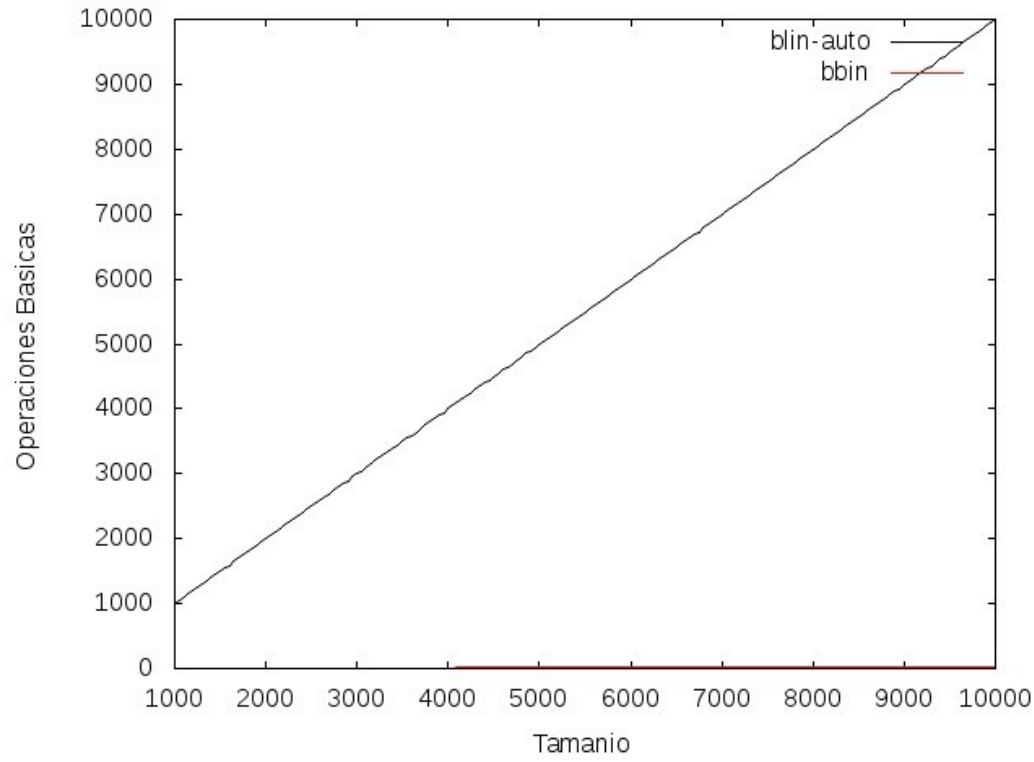
Se puede apreciar que el número medio de OBs va disminuyendo a menudo que aumenta n_veces. Además,a pesar de que en rendimiento medio se observa que bbin es más efectivo,a medida que aumenta n_veces,el rendimiento de blin_auto mejora,acercándose más al de bbin.

Número máximo de OBs entre la búsqueda binaria y la búsqueda lineal auto organizada

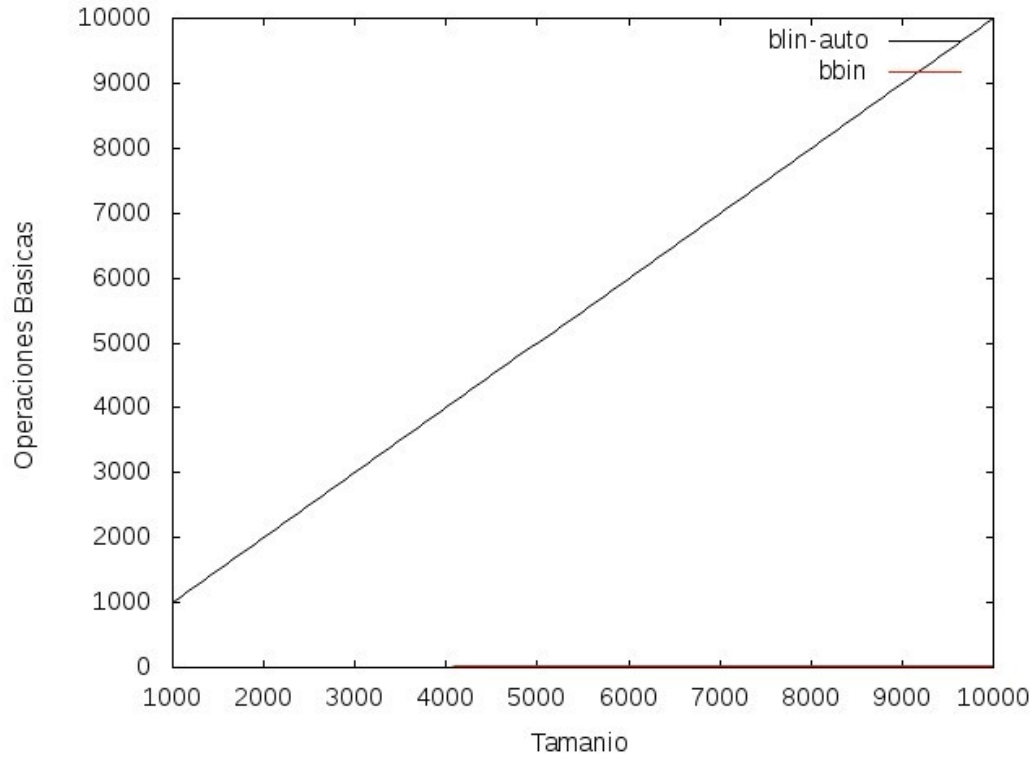
n_veces = 1



n_veces = 100



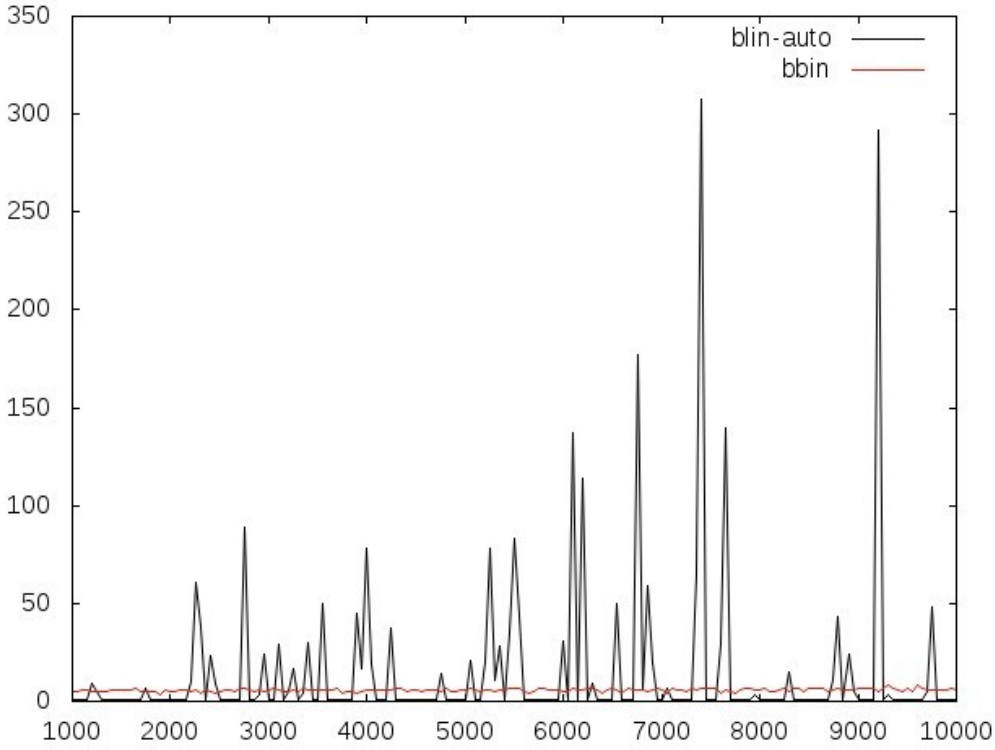
n_veces = 10000



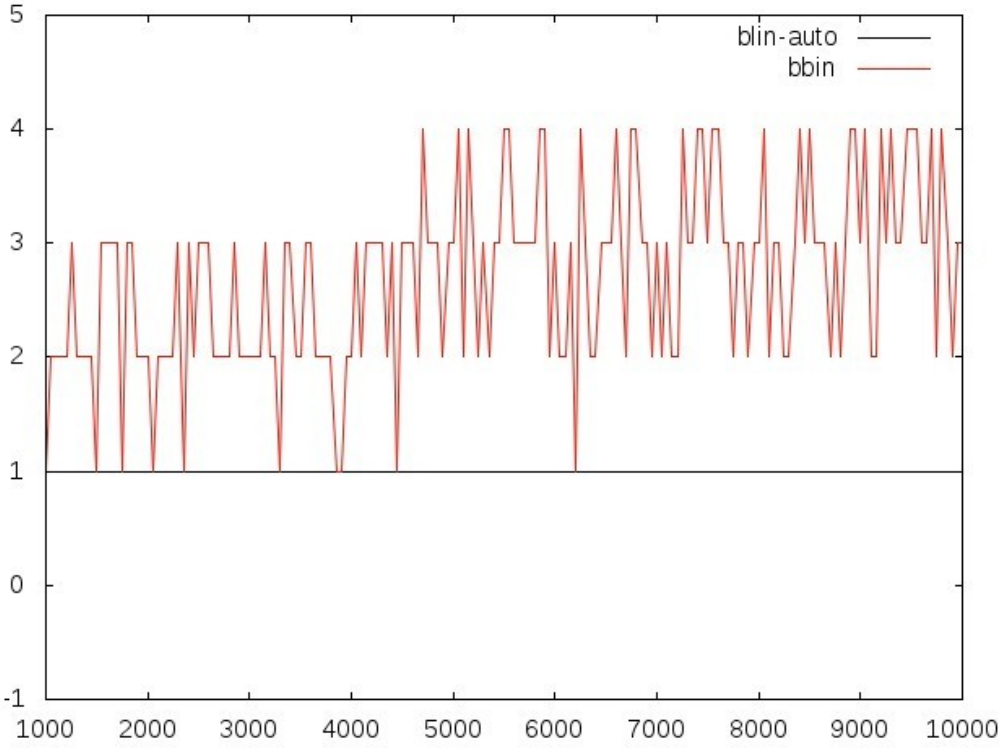
En los tres casos se ve que el número máximo en blin_auto crece linealmente,mientras que en bbin crece pero en valores muy bajos a escala con blin_auto. Se ve así que en el caso peor,bbin es más eficiente que blin_auto

Número mínimo de OBs entre la búsqueda binaria y la búsqueda lineal auto organizada

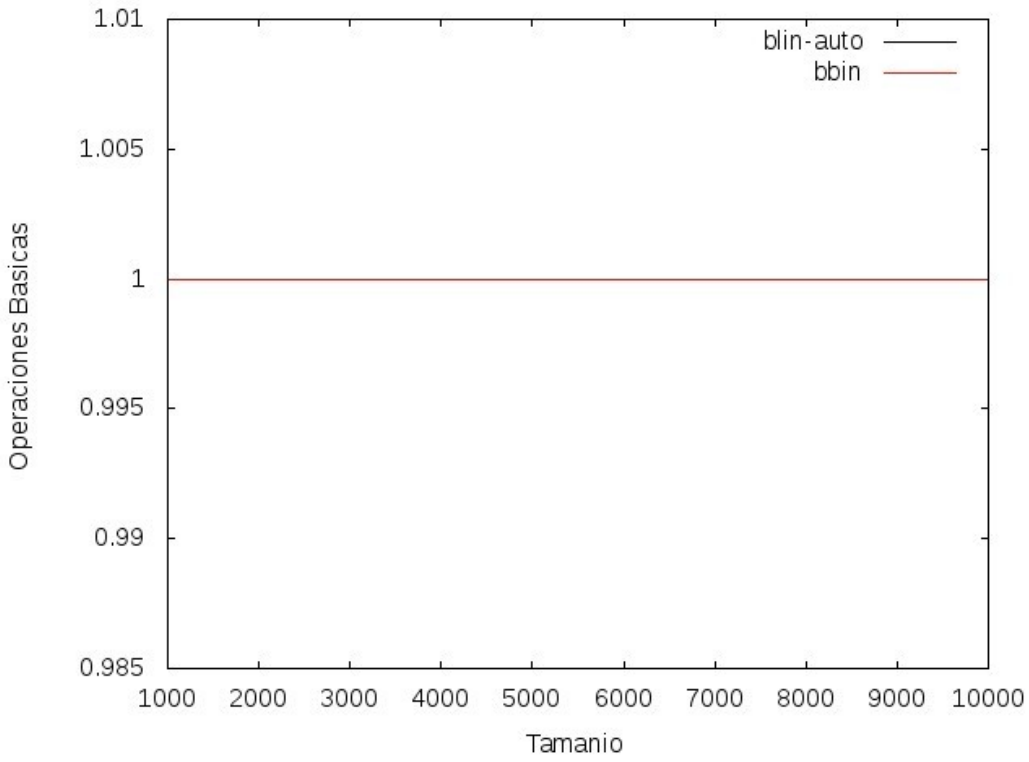
n_veces = 1



n_veces = 100



n_veces = 10000

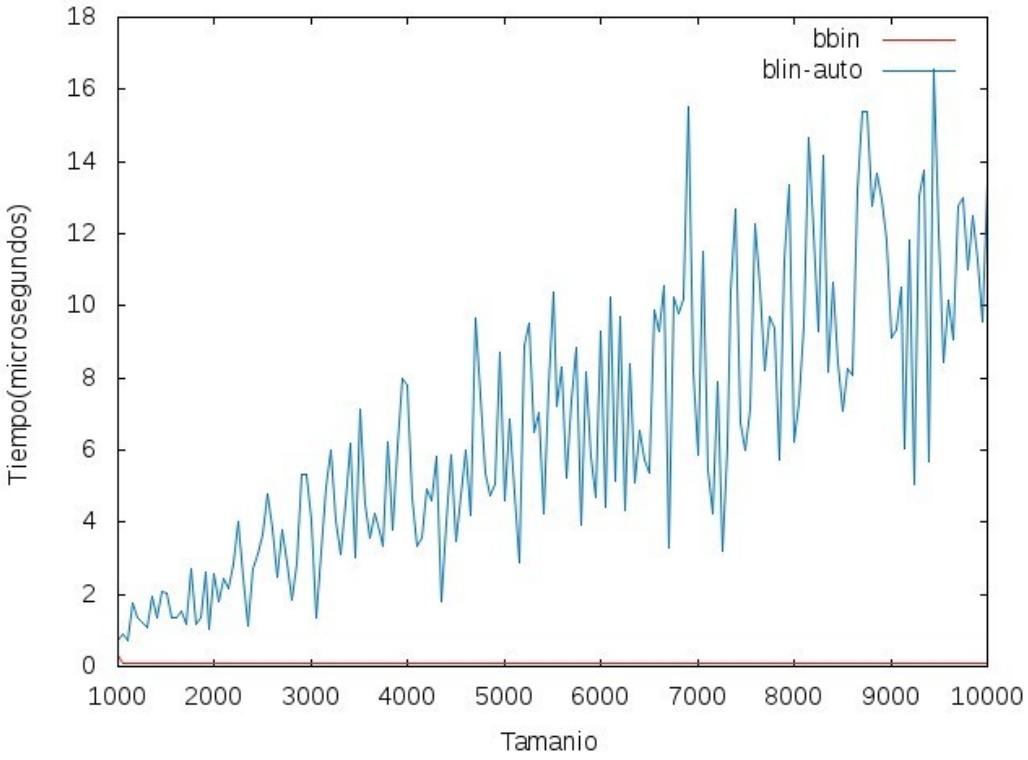


En este caso tenemos 3 gráficas muy distintas. Para n_veces = 1,bbin es mejor ya que blin_auto actuá prácticamente como blin sobre una tabla no ordenada. En n_veces = 100 observamos un cambio,puesto que bbin va variando entre diversos valores mientras que blin_auto se mantiene estable en el 1.Esto se debe a que con tantas búsquedas de claves,y debido a como está implementado blin_auto,las claves más buscadas se van desplazando al principio,de manera que se reduce el coste mínimo al buscarlas. Pero para que esto funcione,n_veces tiene que ser suficientemente grande,como ocurre en este caso.

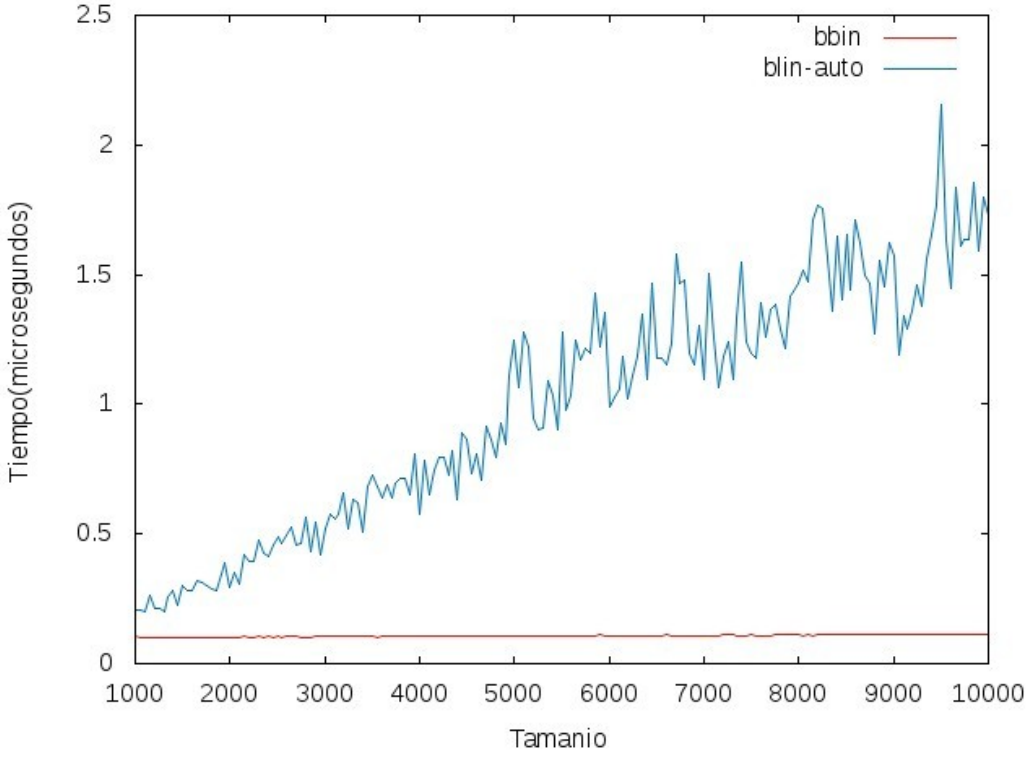
Finalmente,para n_veces = 10000, tanto bbin como blin_auto se mantienen en 1,ya que con tantas búsquedas,la probabilidad de que se de el caso mejor es muy alta,y ocurre para cada tabla. En conclusión,para n_veces suficientemente grandes,el caso mejor de blin_auto es menor o igual que el de bbin.

Tiempo medio de reloj entre la búsqueda binaria y la búsqueda lineal auto organizada

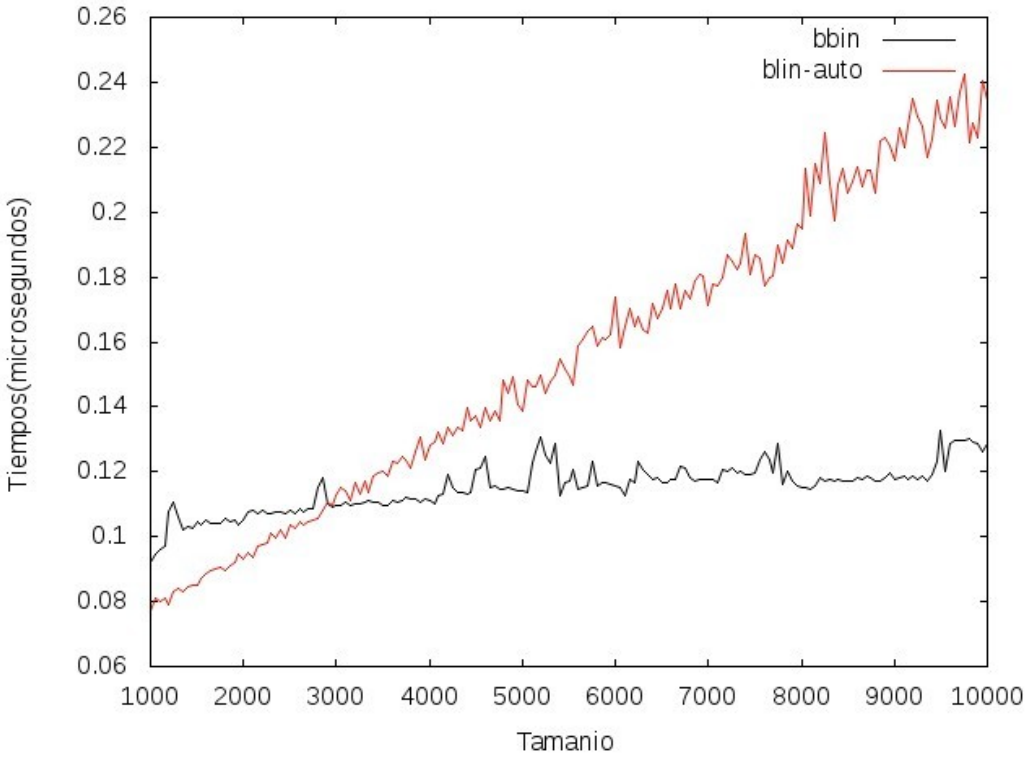
n_veces = 1



n_veces = 100



n_veces = 10000



Al igual que vimos en las OBs promedias,el tiempo de ejecución de bbin es mejor que el de blin_auto,pero a medida que aumenta n_veces,el de blin_auto va mejorando,hasta el punto de que en n_veces = 10000 ,para tamaños entre 1000 y 3000,se ejecuta más rápido que bbin.

5. Respuesta a las preguntas teóricas.

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

En los tres casos,la OB es la comparación entre el elemento de la posición del diccionario en la que se está buscando y la clave a buscar.

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

Wss(1000) de bbin = 10 OBs

Wss(1000) de blin = 1000 OBs

Bss(1000) de bbin = 1 OB

Bss(1000) de blin = 1 OB

5.3 3. Cuando se utiliza blin auto y la distribución no uniforme dada ¿Cómo varía la posición de los elementos de la lista de claves según se van realizando más búsquedas?

Las claves más buscadas se van desplazando hacia el principio de la tabla,de manera que cada vez se requieren menos OBs para encontrarlas.

5.4 ¿Cuál es el orden de ejecución medio de blin auto en función del tamaño de elementos en el diccionario n para el caso de claves con distribución no uniforme dado? Considerar que ya se ha realizado un elevado número de búsquedas y que la lista está en situación más o menos estable.

El caso medio seria aproximadamente 0,005430829(pendiente de la gráfica de OBs medias)*N + O(N)

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

Debido a que la tabla está ordenada,cuando buscas la clave,siempre sabes en que subtabla está,ya sea mayor ,menor o el mismo punto medio de la tabla estudiada. Gracias a esto,se puede ir acotando las posiciones entre las que se encuentra,hasta llegar a la clave buscada.

6. Conclusiones finales.

En esta práctica hemos aprendido y estudiado el rendimiento de distintos algoritmos de búsqueda, además del TAD diccionario. Hemos podido experimentar como la búsqueda más eficiente es bbin,con resultados mucho mejores que blin o blin_auto,y como los diccionarios,con la capacidad de definirlos como ordenados o no ordenados,facilitan mucho la prueba de estos algoritmos.