Análisis de Algoritmos 2019/2020

Práctica 3

Rubén García, Elena Cano.

Código	Memoria	Gráficas	Total

1. Introducción.

La práctica en la que hemos estado trabajando estas últimas semanas tiene como objetivo implementar un diccionario y crear distintos tipos de búsquedas para hallar claves en el. El diccionario emplea como tipo de datos una tabla, sobre las cuales se realizarán dichas búsquedas en las que mediremos el tiempo medio que tardan en realizarse.

2. Objetivos

2.1 Apartado 1

En esta parte de la práctica nos dedicamos principalmente a crear la estructura del diccionario sobre el cual trabajaremos posteriormente. Implementando todas las funciones necesarias para su correcto funcionamiento. Además llevaremos a cabo la creación de tres funciones de búsqueda, cuyo objetivo será encontrar claves dentro del citado diccionario.

2.2 Apartado 2

Una vez implementado el diccionario y las funciones de búsqueda nuestro objetivo en la segunda parte de la práctica será crear funciones mediante las cuales podamos medir el tiempo que tardan en realizar las distintas búsquedas las funciones previamente implementadas.

3. Herramientas y metodología

Para llevar a cabo esta práctica hemos utilizado el entorno de Linux, usando atom para escribir el código y editarlo. Sin embargo debido a los fallos que hemos ido cometiendo hemos necesitado utilizar un depurador, en este caso hemos elegido el de Net Beans. Una vez teníamos el código depurado y funcionando correctamente pasamos Valgrind para detectar así las pérdidas de memoria y corregirlas.

3.1 Apartado 1

La parte principal consiste en implementar correctamente tanto la función del diccionario como las funciones para inicializarlo, liberarlo, insertar todas las claves necesarias y realizar búsquedas. Además programaremos las funciones para llevar a cabo búsquedas lineales, lineales autoorganizadas y binarias.

Cabe destacar la implementación de la función blin_auto pues esta además de encontrar la clave se intercambia con la posición anterior, excepto si la clave encontrada ya está en la primera posición de la tabla.

Todas ellas tiene la función de devolver la posición en la que se encuentra la clave, o no encontrado en caso de que no esté, por medio de la posición pasa como argumento ‘ppos’. Además se imprimirá el número de operaciones básicas realizadas por cada una de ellas.

3.2 Apartado 2

En este caso hemos llevado a cabo la implementación de dos nuevas funciones las cuales son: tiempo_medio_busqueda y genera_tiempos_busqueda. El principal objetivo llevado a cabo por ambas es guardar en nuestra estructura ptiempo los numeros minimos, máximos y medio de operaciones básicas realizadas y el tiempo medio de ejecución.

Para tiempo_medio_busqueda nos tenemos que crear un diccionario, y una permutación de tamaño N mediante la rutina genera_perm. Insertamos todos los elementos que acabamos de crear mediante una función implementada en el primer apartado, en este caso es: insercion_masiva_diccionario. Nos hemos creado una tabla en la que guardaremos las claves que queremos buscar y estas las generamos mediante una función pasada como argumento. Finalmente medimos el tiempo que se tarda en encontrar todas las claves yna vez recogidos todos los datos oportunos los guardaremos en la estructura ptiempo. Liberamos memoria y se sale de la función.

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)
{
  PDICC pdicc=NULL;
  if(tamanio<0 || (orden!=ORDENADO && orden!=NO_ORDENADO)){
    return NULL;
  }
    pdicc=malloc(sizeof(PDICC));
    if(pdicc==NULL)
      return NULL;

    pdicc->tabla=malloc(tamanio*sizeof(int));
    if (pdicc->tabla==NULL){
      free(pdicc);
      return NULL;
    }
    pdicc->tamanio=tamanio;
    pdicc->n_datos=0;
    pdicc->orden=orden;

    return pdicc;
}

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

int inserta_diccionario(PDICC pdicc, int clave)
{
  int i, OBS=0;

  if(pdicc==NULL)
    return ERR;
  if (pdicc->n_datos>=pdicc->tamanio)
    return ERR;

  pdicc->tabla[pdicc->n_datos]=clave;

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

```
  }
  pdicc->n_datos++;

  return OBS;
}

int insercion_masiva_diccionario (PDICC pdicc, int *claves, int
n_claves)
{
  int i, flag, OBS=0;
  if (pdicc==NULL || claves==NULL || n_claves<0)
    return ERR;
  for (i=0; i<n_claves; i++) {
    flag=inserta_diccionario(pdicc, claves[i]);
    if (flag==ERR)
      return ERR;
    OBS+=flag;
  }
  return OK;
}

int busca_diccionario(PDICC pdicc, int clave, int *ppos,
pfunc_busqueda metodo)
{
  int OBS=0;

  if (pdicc==NULL || ppos==NULL || metodo==NULL)
    return ERR;
  OBS=metodo(pdicc->tabla, 0, pdicc->n_datos-1, clave, ppos);
  if (OBS==ERR)
    return ERR;
  return OBS;
}
int bbin(int *tabla, int P, int U, int clave, int *ppos)
{
  int medio, OBS=0;

      if (tabla==NULL || P>U)
    return ERR;

  while(P<=U) {
    medio=(P+U)/2;
    OBS++;
    if (tabla[medio]==clave){
      *ppos=medio;
      return OBS;
    }
    else if (clave<tabla[medio]) {
      U=medio-1;
    }
    else if (clave>tabla[medio]) {
      P=medio+1;
    }
  }

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

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

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

4.2 Apartado 2

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

  PDICC pdicc;
  int *claves=NULL;
  int *perm=NULL;
  int pos, obs_sol, j;
  double t1, t2;

  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;
  }

  claves=malloc(N*n_veces*sizeof(int));
  if (claves==NULL) {
    libera_diccionario(pdicc);
    free(perm);
    return ERR;
  }

  generador(claves, (N*n_veces), N); /*o exponencial dependiendo
del algoritmo de ordenacion q usemos*/

  t1=clock();
  if (t1==-1) {
    libera_diccionario(pdicc);
    free(perm);
    free(claves);
    return ERR;
  }

  ptiempo->N=N;
  ptiempo->n_elems=N*n_veces;
  ptiempo->tiempo=0;
  ptiempo->medio_ob=0;
  ptiempo->min_ob=INT_MAX;
  ptiempo->max_ob=-1;

  for(j=0;j<=N*n_veces;j++) {
    obs_sol=busca_diccionario(pdicc, claves[j],&pos,metodo);
    if (obs_sol==ERR){
      libera_diccionario(pdicc);
      free(perm);
      free(claves);
      return ERR;
    }
    if (obs_sol<ptiempo->min_ob)
      ptiempo->min_ob=obs_sol;
    if (obs_sol>ptiempo->max_ob)
      ptiempo->max_ob=obs_sol;
    ptiempo->medio_ob+=((double)obs_sol/(n_veces*N));
  }

  t2=clock();
  if (t2==-1) {
    libera_diccionario(pdicc);
    free(perm);
    free(claves);
    return ERR;
  }
```

```
  ptiempo->tiempo=((double)(t2-t1)/(n_veces*N));

  libera_diccionario(pdicc);
  free(perm);
  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) {
  PTIEMPO tiempos=NULL;
  int i, counter=0;

  if (metodo==NULL || generador==NULL || fichero==NULL || num_min<0
|| num_max<num_min || incr<1)
    return ERR;

  tiempos=malloc((((num_max-num_min)/incr)+1)*sizeof(TIEMPO));
  if (tiempos==NULL)
    return ERR;

  for(i=num_min; i<=num_max; i+=incr, counter++) {
    if (tiempo_medio_busqueda(metodo, generador, orden, i, n_veces,
&tiempos[counter])==ERR) {
      free(tiempos);
      return ERR;
    }
  }

  if(guarda_tabla_tiempos(fichero, tiempos, counter)==ERR) {
    free(tiempos);
    return ERR;
  }

  free(tiempos);
  return OK;

}
```

5. Resultados, Gráficas

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

5.1 Apartado 1

Resultados del apartado 1.

Modificamos el programa ejercicio1 para comprobar el correcto funcionamiento de bbin y blin en una tabla ordenada de 10 elementos, ejecutamos primero para bbin y, a continuación para blin, dando como resultado las siguientes salidas en la terminal usando como clave el número 7:

- Para bbin:

  Clave 7 encontrada en la posicion 6 en 4 op. basicas

  1 2 3 4 5 6 7 8 9 10

- Para blin:

  Clave 7 encontrada en la posicion 6 en 7 op. basicas

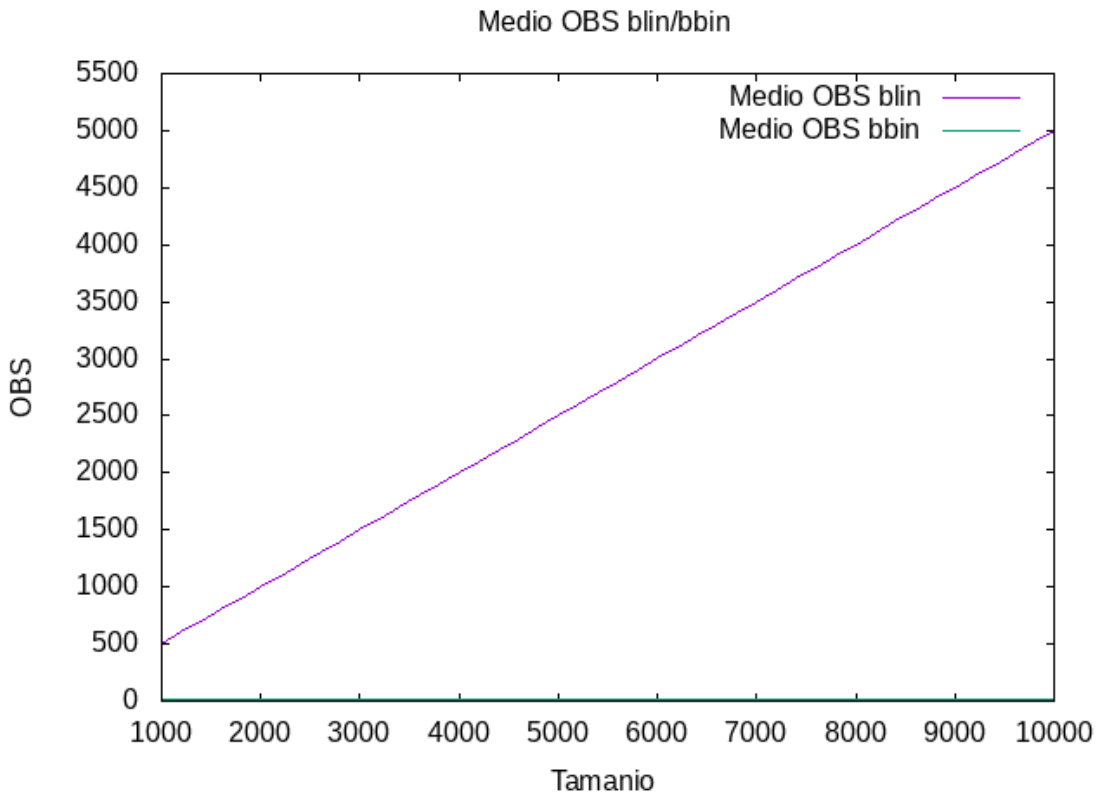
  1 2 3 4 5 6 7 8 9 10

Podemos ver como en ambos casos la búsqueda funciona de manera correcta, en ambos casos devuelve correctamente la posición de la clave 7 (que es 6 debido a que empieza a contar en la posición 0). Se puede apreciar también como el número de operaciones básicas de bbin es menor que blin y que coincide con lo estudiado en teoría tanto para bbin como para blin.

5.2 Apartado 2

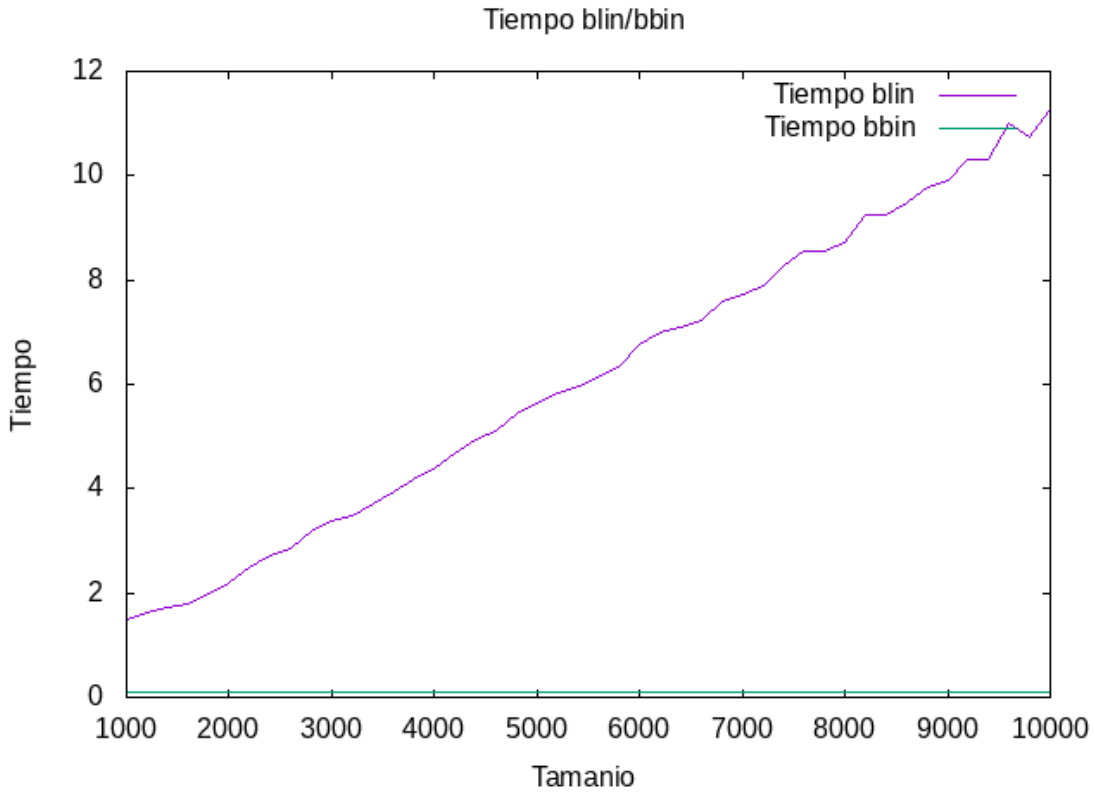
Resultados del apartado 2.

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



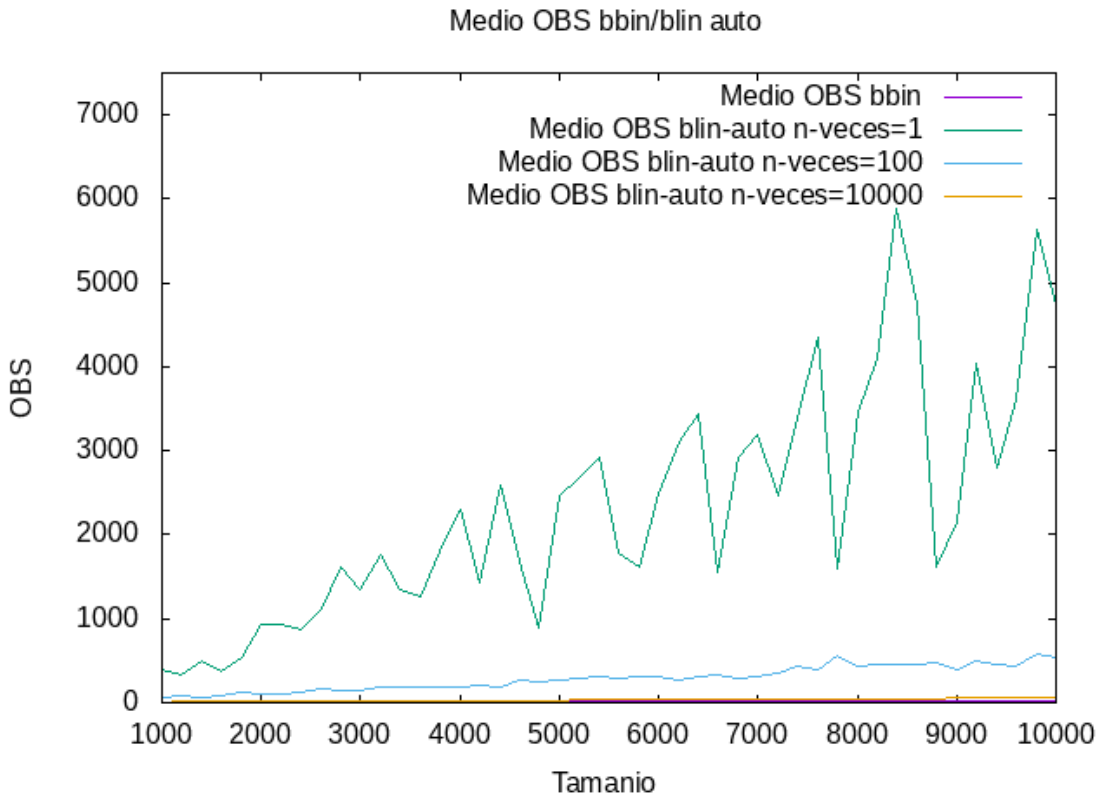
Se puede apreciar como aparentemente Medio OBS bbin es la línea del 0, esto es debido a que en comparación a blin, ésta última necesita muchas más operaciones básicas de media. Esto se puede ver fácilmente viendo como, en el peor de los casos, $A_{Bbin}=\Omega(\lg(n))$, mientras que $A_{Blin}(n)=n/2+O(1)$, por lo que Blin es mucho mayor en comparación con Bbin.

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



De nuevo, parece como tiempo bbin es la línea del 0, sin embargo, y como hemos explicado antes esto es debido a que el tiempo promedio que tarda la búsqueda lineal en ser efectiva es considerablemente mayor que la búsqueda binaria. Sabemos además que la búsqueda binaria es óptima para los tres casos (mejor, medio y peor).

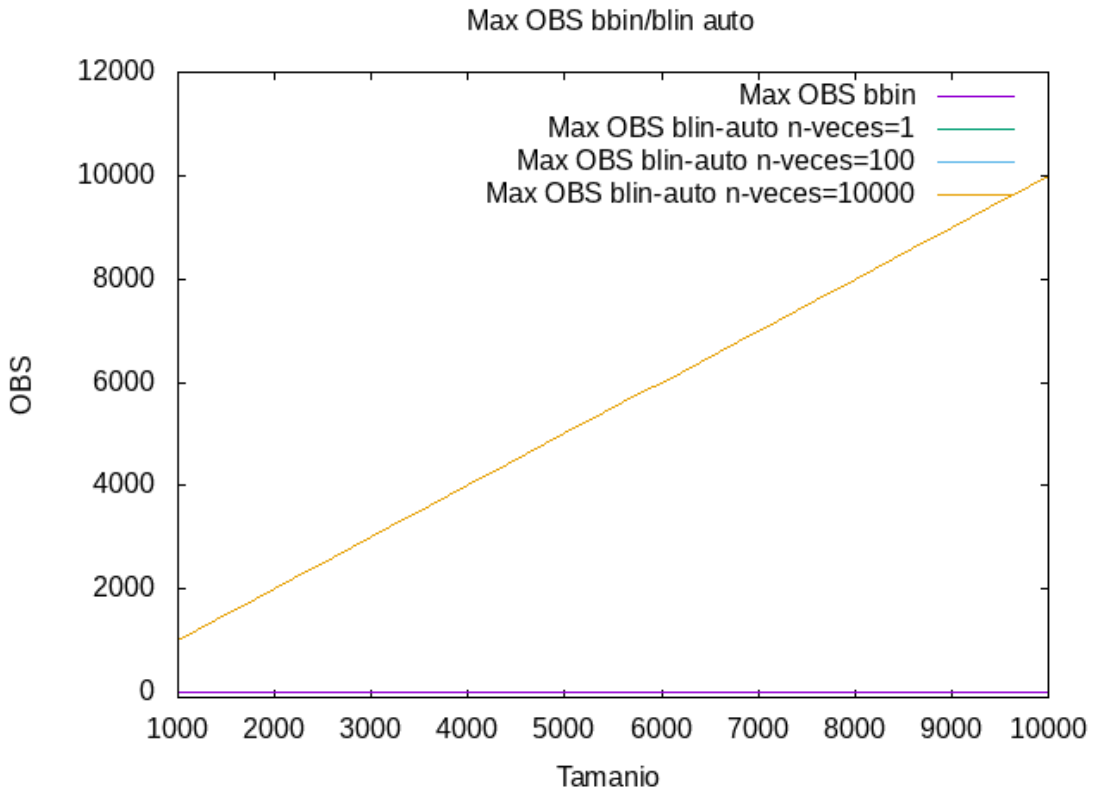
Gráfica comparando el número promedio de OBs entre la búsqueda binaria y la búsqueda lineal auto organizada (para los valores de n_veces=1, 100 y 10000), comentarios a la gráfica.



Para el número medio de OBS realizadas por blin-auto podemos observar que a mayor número de veces ejecutado menor son las operaciones básicas que tiene que realiza. Esto se debe principalmente a que cuanto más se busca un número, más puestos avanza hacia delante. Por lo tanto al volver a buscar ese mismo elemento el tiempo de ejecución se reduce. Al buscarlo una única vez el elemento se puede encontrar en cualquier posición, sin embargo al realizar más búsquedas los elementos que aparecen más veces generados potencialmente van a aparecer al principio de la tabla, esta es la razón por la cual el número de operaciones básicas se reduce en función de cuántas veces ejecutamos nuestro algoritmo.

La gráfica del caso medio de bbin a penas se puede apreciar, debido a la escala parece una recta, sin embargo representa log(n).

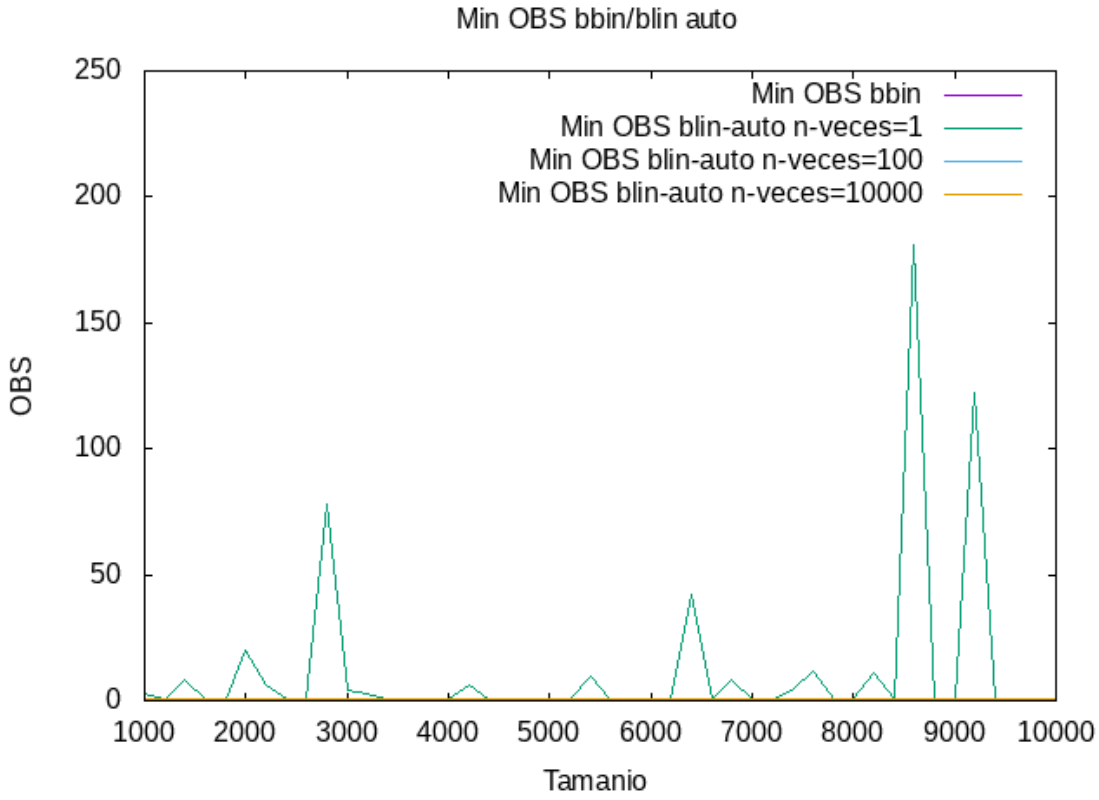
Gráfica comparando el número máximo de OBs entre la búsqueda binaria y la búsqueda lineal auto organizada (para los valores de n_veces=1, 100 y 10000), comentarios a la gráfica.



En la gráfica podemos ver una única pendiente la cual representa el caso peor de blin auto, ya sea para 1, 100, o 10000 veces ejecutado el algoritmo. Esto se debe a que para una tabla de n elementos al buscar el último de la tabla siempre dará n operaciones básicas, dando igual que el algoritmo mueva el número una posición hacia delante, pues si volvemos a buscar el último número las OBS no cambian.

La recta de abajo en realidad está representando log(n) que es el caso peor de bbin, sin embargo al tener que representarlo en una escala tan grande no se puede apreciar con claridad.

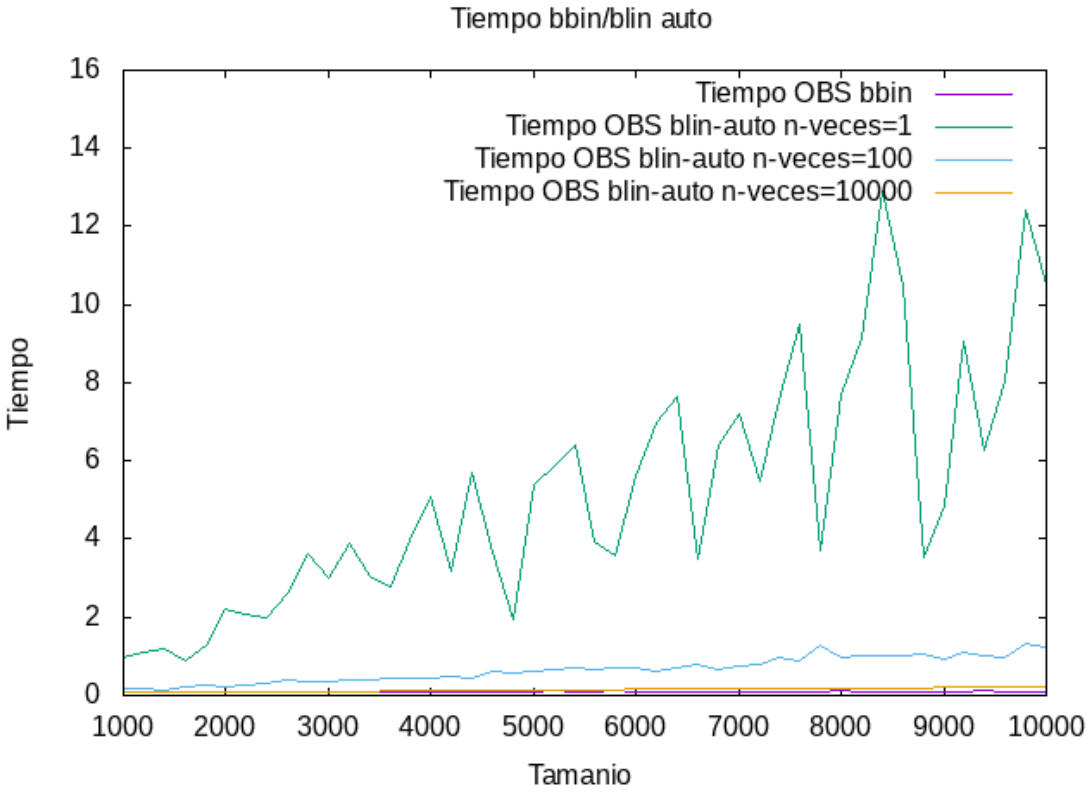
Gráfica comparando el número mínimo de OBs entre la búsqueda binaria y la búsqueda lineal auto organizada (para los valores de n_veces=1, 100 y 10000), comentarios a la gráfica.



Podemos ver que cuando blin auto se ejecuta una única vez el mínimo de operaciones básicas nos va a dar algunos picos, esto se debe a que al buscar una clave la lista está desordenada y puede encontrarse en cualquier posición. Sin embargo cuando ejecutamos el algoritmo 100 o 10000 veces las claves cada vez avanzan más posiciones y el mínimo de operaciones básicas se reduce más hasta llegar a prácticamente a 1.

La gráfica para el caso mejor de bbin es una recta en 1.

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.



Como podemos observar el tiempo para blin auto ejecutado una única vez es mucho mayor que cuando ha sido ejecutado 100 o 10000 veces el cual se reduce debido a que los elementos más buscados de la lista comienzan a aparecer en la primeras posiciones.

6. Respuesta a las preguntas teóricas.

6.1 Pregunta 1

En los tres casos funciona por comparación de clave, la operación básica de bbin, blin y blin_auto es tabla[i]==clave siendo i la posición de la tabla en la que nos encontramos en dicho momento.

6.2 Pregunta 2

Para blin el mejor de los casos es que la clave se encuentre en el primer elemento dentro del diccionario y el peor que se encuentre el último, así que $W_{Blin}(n)=n$, $B_{Blin}(n)=O(1)$.

Para bbin el mejor de los casos es que la clave se encuentre en el medio del diccionario, mientras que el peor de los casos (siguiendo lo estudiado en teoría y teniendo en cuenta que bbin funciona de manera binaria) es $W_{Bbin}(n)=\lceil lg(n) \rceil=lg(n)+O(1)$, por lo que tenemos que $W_{Bbin}(n)=\Omega(lg(n))$, $B_{Bbin}(n)=O(1)$.

6.3 Pregunta 3

El algoritmo de búsqueda blin auto cada vez que se realiza la búsqueda de una clave ésta intercambia su posición con la anterior, excepto si la clave encontrada ya está en la primera posición de la tabla. Por lo tanto cuantas más veces se realice la búsqueda de una clave esta irá avanzando posiciones y el número de operaciones básicas realizadas para encontrarla irá disminuyendo. Puede incluso llegar a un punto en el que la clave se encuentre en primera posición y las obs realizadas serán el número mínimo.

6.4 Pregunta 4

El porcentaje de aparición una clave en la búsqueda lineal autoorganizada depende del número de consultas que hagamos a esa clave, pues se va "ordenando" según se van realizando estas consultas.

En un primer inicio, la tabla está desordenada, sin embargo hay claves que aparecen con más frecuencia que otras, el caso medio de la búsqueda lineal autoorganizada siguiendo la ley de Zipf (distribución geométrica que da como resultado una probabilidad logarítmica) es la siguiente:

$$C_N = \sum_{i=1}^{N} \frac{log(i)}{i} \text{ y } S_N = \sum_{i=1}^{N} \frac{i*log(i)}{i} = \sum_{i=1}^{N} log(i)$$

Una vez aproximado por integrales (como se estudió en teoría), se obtiene:

$C_N \sim \frac{1}{2}\left(log(N)\right)^2$ y $S_N \sim Nlog(N)$, y por lo tanto: $A_{BLin}(N) \sim \frac{2N}{log(N)}$.

Sin embargo, a medida que se realizan muchas consultas, la tabla se "ordena" y el caso medio de la búsqueda lineal autoorganizada y cambia y se vuelve más eficiente, veamos su funcionamiento. Una aproximación sencilla es (aplicando la ley 80/20) suponer que cuando se busca una clave probable (dentro del 80% de probabilidad) se encuentra igualmente distribuida en el primer 20% del diccionario, y aplicamos lo mismo a las claves poco probables. Con esta aproximación podemos estimar el caso medio de la búsqueda lineal autoorganizada de la siguiente manera:

$$A_{BLin}(N) = 0{,}8 * \left(\frac{0{,}2N}{2}\right) + 0{,}2 * \left(\frac{N-0{,}2N}{2}\right) = 0{,}8 * 0{,}2N + 0{,}2 * 0{,}8N$$

Esto queda como resultado $A_{BLin}(N) \approx 0{,}16N$. Otros análisis más complejos, como el llevado a cabo por Donald Knuth siguiendo la ley de Zipf establecen que el caso medio da como resultado $A_{BLin}(N) \approx 0{,}122N$ y que, siguiendo otras distribuciones mejores aproximadas que la ley de Zipf el coste medio es $A_{BLin}(N) = O(1)$ según el número de claves del diccionario se aproxima a infinito. En cualquier caso podemos apreciar como el caso medio es sustancialmente mejor, $A_{BLin-auto}(N) = 0{,}16 * N$ (en el caso peor aproximado) frente a $A_{BLin}(N) = 0{,}5 * N + O(1)$.

6.5 Pregunta 5

El rendimiento de búsqueda binaria es óptima pues puede ser analizada como un árbol binario de búsqueda, donde la raíz es el elemento medio del diccionario, y los hijos izquierdo y derecho son respectivamente el elemento medio de la primera parte y el elemento medio de la segunda parte, y así sucesivamente.

El rendimiento de los árboles binarios de búsqueda es muy eficiente, pues representan una función logarítmica. En concreto sabemos que la búsqueda binaria es óptima para el caso peor, medio y mejor, siendo estos $W_{Bbin}(n)=\Omega(lg(n))$, $A_{Bbin}(n)=\Omega(lg(n))$, $B_{Bbin}(n)=O(1)$.

7. Conclusiones finales.

Al llevar a cabo la práctica nos ha permitido comprender cómo funciona la estructura de diccionario y cómo trabajar con ella para insertar y buscar distintas claves. Además hemos descubierto una nueva función de búsqueda, el algoritmo de búsqueda lineal autoorganizada.

Hemos podido comprobar experimentalmente las casos mejor, peor y medio de las distintas funciones y cuanto tardan éstas encontrar un cierto número de claves.