Análisis de Algoritmos 2020/2021

Práctica 3

Daniel Cerrato y David Garitagoitia, Grupo 1261.

1. Introducción.

En esta práctica, se van a poner en uso las nociones que tenemos sobre los algoritmos de búsqueda, con los que, sumando los conocimientos anteriores, generaremos tablas de tiempos y operaciones básicas para compararlos.

2. Código

2.1 Archivo "busqueda.c" (Archivo con las funciones necesarias para manejar el TAD Diccionario y con las funciones de búsqueda)

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

  PDICC dic = NULL;

  dic = (PDICC)malloc(sizeof(DICC));

  if (!(dic->tabla = (int *)malloc(tamanio * sizeof(int))))

    return NULL;

  dic->orden = orden;

  dic->tamanio = tamanio;

  dic->n_datos = 0;

  return dic;

}



void libera_diccionario(PDICC pdicc){

  free(pdicc->tabla);

  free(pdicc);

}
```

```
int inserta_diccionario(PDICC pdicc, int clave){
  int aux;
  if (!pdicc) return 0;
  switch (pdicc->orden)  {
  case ORDENADO:
    if (pdicc->n_datos == pdicc->tamanio) return ERR;
    aux = pdicc->n_datos;
    while (aux != 0 && pdicc->tabla[aux - 1] > clave){
      pdicc->tabla[aux] = pdicc->tabla[aux - 1];
      aux--;
    }
    pdicc->tabla[aux] = clave;
    pdicc->n_datos++;
    return pdicc->n_datos - aux;
  default:
    if (pdicc->n_datos == pdicc->tamanio) return ERR;
    pdicc->tabla[(pdicc->n_datos)++] = clave;
    return 1;
  }
}


int busca_diccionario(PDICC pdicc, int clave, int *ppos, pfunc_busqueda metodo){
  if (!pdicc || !ppos || !metodo) return ERR;
  return metodo(pdicc->tabla, 0, pdicc->n_datos, clave, ppos);
}
```

```
int insercion_masiva_diccionario(PDICC pdicc, int *claves, int n_claves){
  int i, ob, OBs = 0;
  for (i = 0; i < n_claves; i++) {
    if ((ob = inserta_diccionario(pdicc, claves[i])) == ERR) return ob;
    OBs += ob;
  }
  return OBs;
}

int bbin(int *tabla, int P, int U, int clave, int *ppos){
  int aux = 0, m = (P + U) / 2;
  if (P > U){
    (*ppos) = NO_ENCONTRADO;
    return ERR;
  }
  if (tabla[m] == clave){
    (*ppos) = m;
    return 2;
  }
  if (tabla[m] > clave){
    if ((aux = bbin(tabla, P, --m, clave, ppos)) == ERR) return ERR;
    return (3 + aux);
  }
  if ((aux = bbin(tabla, ++m, U, clave, ppos)) == ERR) return ERR;
  return (3 + aux);
}
```

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

  if (!tabla || !ppos || P < 0) return ERR;

  for ((*ppos) = P; (*ppos) != U; (*ppos)++){

    if (tabla[(*ppos)] == clave) return (*ppos) - P + 1; /*tantas comparaciones de clave como
elementos entre el primero y la posición donde lo encontraste*/

  }

  (*ppos) = NO_ENCONTRADO;

  return U-P;

}



int blin_auto(int *tabla, int P, int U, int clave, int *ppos){

  int aux = 0;

  aux = blin(tabla, P, U, clave, ppos);

  if (*ppos != NO_ENCONTRADO && *ppos != 0) swap(&tabla[(*ppos)], &tabla[(*ppos) - 1]);

  return aux;

}
```

2.2 Archivo “tiempos.c” (Archivo que contiene las funciones para generar y guardar tablas de tiempos usando los algoritmos de búsqueda. Además, contiene las funciones antiguas de generación de tiempos de los algoritmos de ordenación)

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

  PDICC dic = NULL;

  int *claves = NULL, ppos, aux = n_veces * N;

  clock_t ini, end;

  long int i, ob, obs = 0, ob_min = INT_MAX, ob_max = 0;

  if (!(dic = ini_diccionario(N, orden))) return ERR;
```

```
if ((claves = genera_perm(N)) == NULL) {
  libera_memoria(claves, dic);
  return ERR;
}
insercion_masiva_diccionario(dic, claves, N);
free(claves);
claves = (int *)malloc(aux * sizeof(int));
if (claves == NULL){
  libera_memoria(claves, dic);
  return ERR;
}
generador(claves, aux, N);
if (claves == NULL){
  libera_memoria(claves, dic);
  return ERR;
}
ini = clock();
for (i = 0; i < aux; i++){
  ob = busca_diccionario(dic, claves[i], &ppos, metodo);
  if (ob == ERR){
    libera_memoria(claves, dic);
    return ERR;
  }
  obs += ob;
  if (ob < ob_min) ob_min = ob;
```

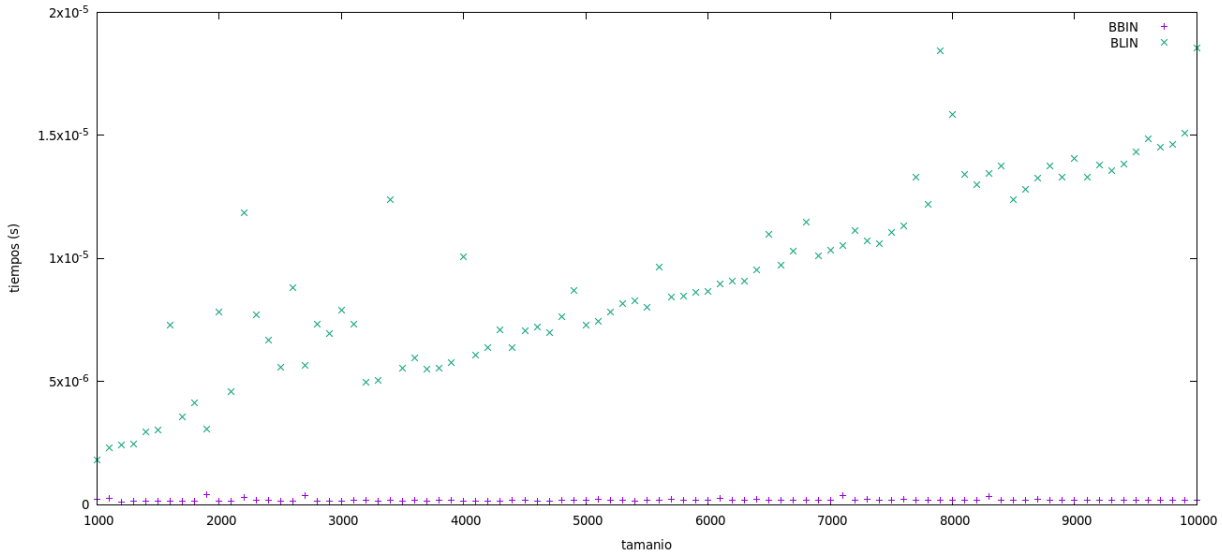
```
    else if (ob > ob_max) ob_max = ob;
  }
  end = clock();
  ptiempo->N = N;
  ptiempo->n_elems = aux; /
  ptiempo->tiempo = (double)(end - ini) / aux / CLOCKS_PER_SEC;
  ptiempo->medio_ob = (double)obs / aux;
  ptiempo->min_ob = ob_min;
  ptiempo->max_ob = ob_max;
  libera_memoria(claves, dic);
  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 t = NULL;
  int i = 0, n_tiempos, tamanio = num_min;
  if (num_min > num_max) return ERR;
  n_tiempos = (num_max - num_min) / incr + 1;
  if ((t = (PTIEMPO)malloc(n_tiempos * sizeof(t[0]))) == NULL) return ERR;
  for (i = 0; i != n_tiempos; i++){
    if (tiempo_medio_busqueda(metodo, generador, orden, n_veces, tamanio, &t[i]) ==
ERR){
      free(t);
      return ERR;
    }
```

```
    tamanio += incr;
  }
  guarda_tabla_tiempos(fichero, t, n_tiempos);
  free(t);
  return OK;
}
```

3. Comparaciones

Inicialmente, se compararán los tiempos y operaciones básicas medias de los algoritmos de búsqueda lineal y binaria. Para ello se irán buscando claves (1 vez por cada una) en listas de números que van desde los 1000 hasta los 10000 elementos.

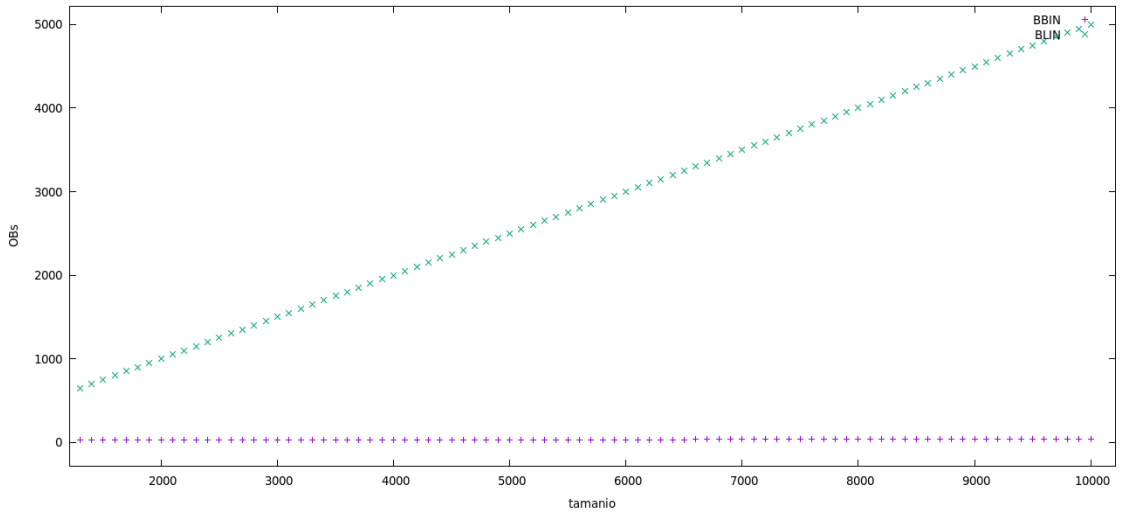
Para esta parte, cuando se use búsqueda lineal, la lista no estará ordenada; sin embargo, para búsqueda binaria, si lo estará.



Como se puede apreciar, los tiempos de búsqueda binaria son muy bajos en relación con los de búsqueda lineal. Esto se debe a que los tiempos medios de la búsqueda binaria se acercan a lg(N), mientras que los de búsqueda lineal se disparan en forma de O(N).

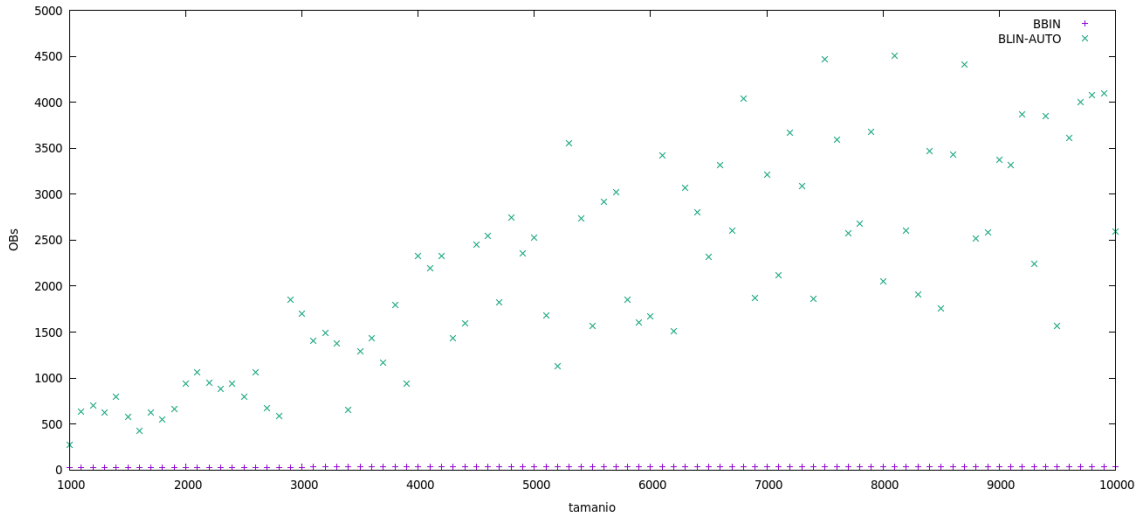
Más tarde, veremos como esa recta rosada referente a bbin, deja de parecer graficar f(x)=0 y se convierte en una gráfica muy parecida a la del logaritmo binario.

En la siguiente gráfica podemos observar el gran parecido con la anterior, pero esta vez estamos hablando de OBs medias. La gráfica de BLin, vuelve a representar una función típica con coste O(N), en este caso N/2, mientras que BBin parece ser 0 siempre, sin embargo, veremos que realmente es una función muy baja en coste, de orden O(lg(N)).

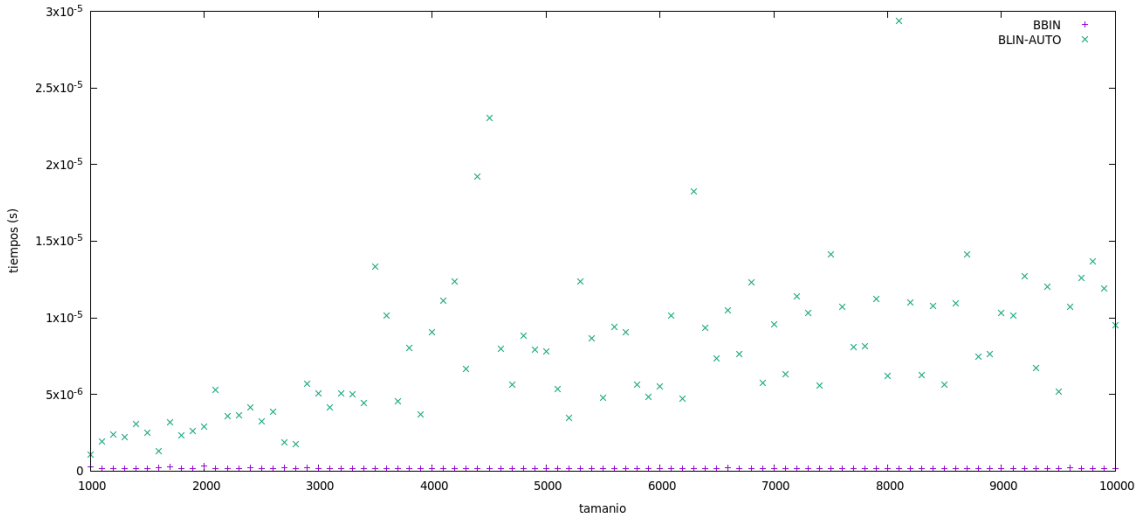


Seguidamente compararemos los algoritmos de búsqueda lineal auto-organizada y búsqueda binaria, para ver si es capaz de hacerle algo más de frente.

En este caso los tamaños de las listas seguirán siendo los mismos, sin embargo, las claves se buscarán 1, 100 y 10000 veces. Para este caso, se generarán una lista de claves a buscar de forma potencial, es decir, algunas claves se buscarán más que otras.

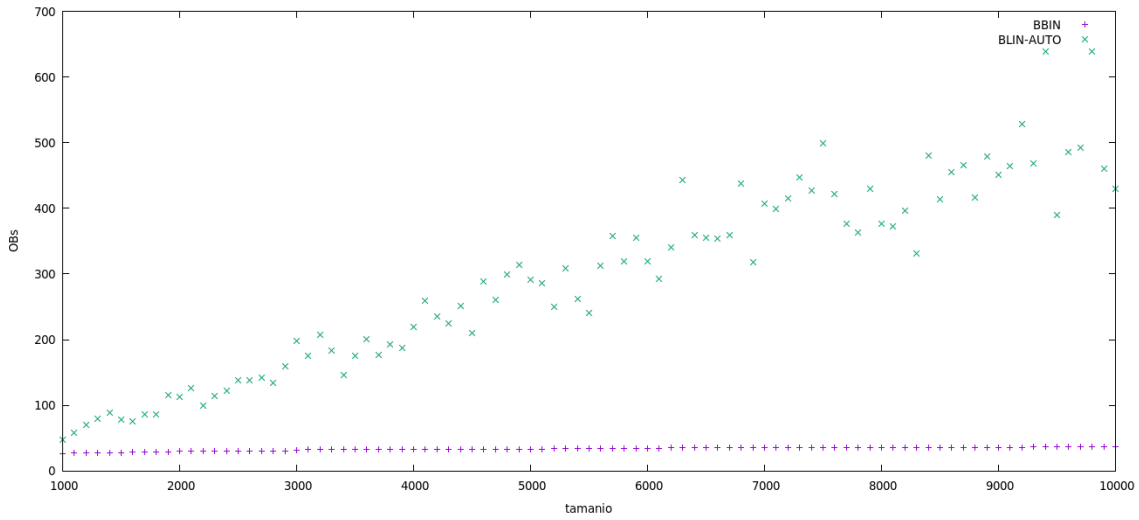


Estamos en el caso 1, donde n_veces = 1. A pesar de esto, como estamos usando el generador de claves potencial, puede que alguna clave se busque varias veces y que alguna otra no se busque. Es por esto que la gráfica de BLin_auto es tan dispar y no es lineal como pasaba con BLin. Aun así, se puede observar el crecimiento a medida que aumenta el tamaño del diccionario. BBin sigue siendo tan eficiente como hasta el momento, con esa apariencia de constancia en 0 que le brinda la escala.

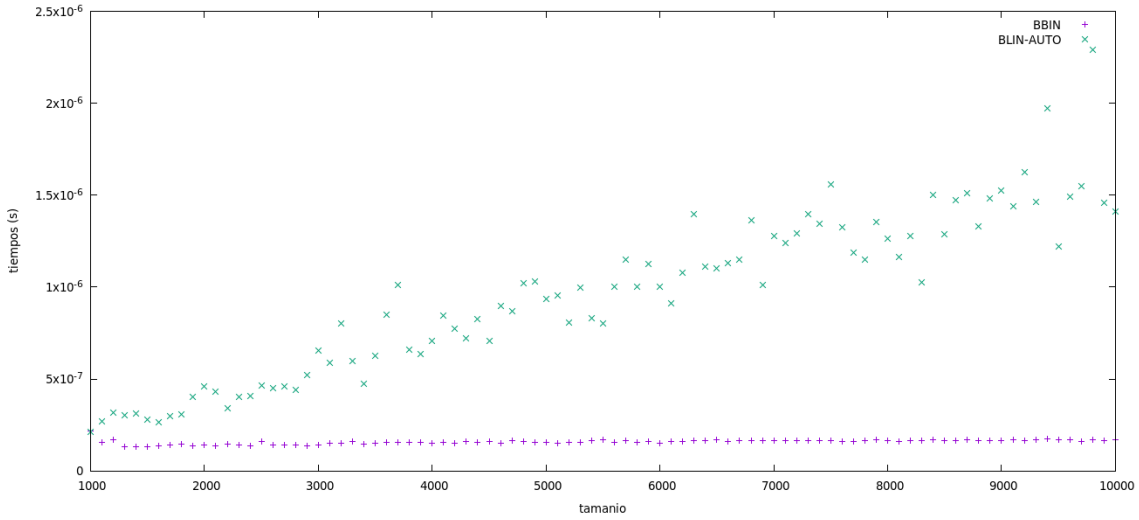


En cuanto a la gráfica de tiempos, seguimos en las mismas, aunque esta vez el crecimiento de los tiempos para BLin_auto, es mucho más suave que para BLin. Aun así, seguimos viendo tiempos muy dispares promovidos por la generación potencial de claves, que sumado a que no se están buscando muchas veces la mismas claves, su función real no se aprecia en los tiempos ni OBs. BBin sigue en su norma.

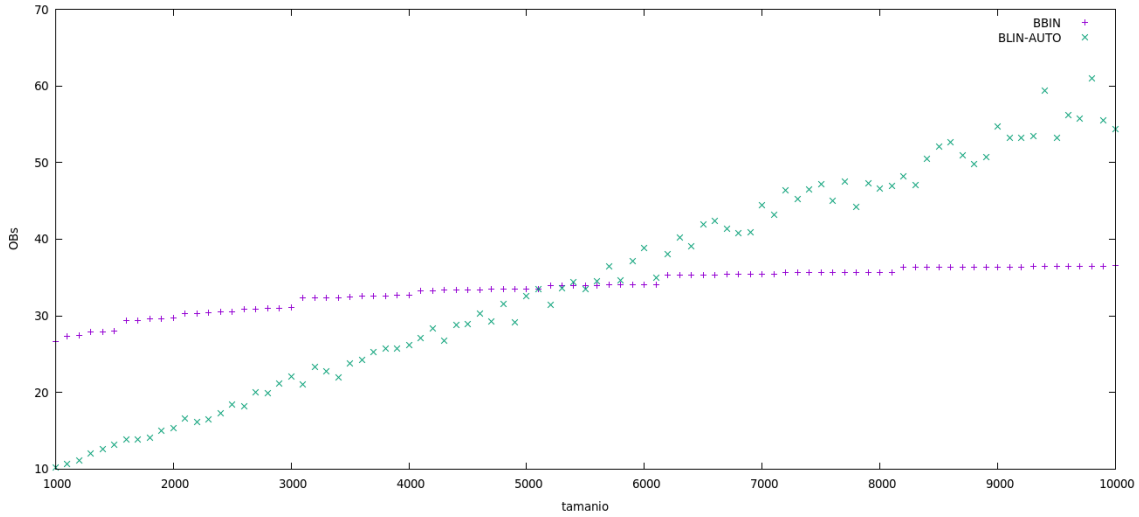
Entramos en el caso 2, donde n_veces = 100. Es decir, ahora se buscarán 100 veces más claves que antes. Esto permite que las claves se repitan más veces individualmente y veremos que BLin_auto empieza a dar sus frutos.



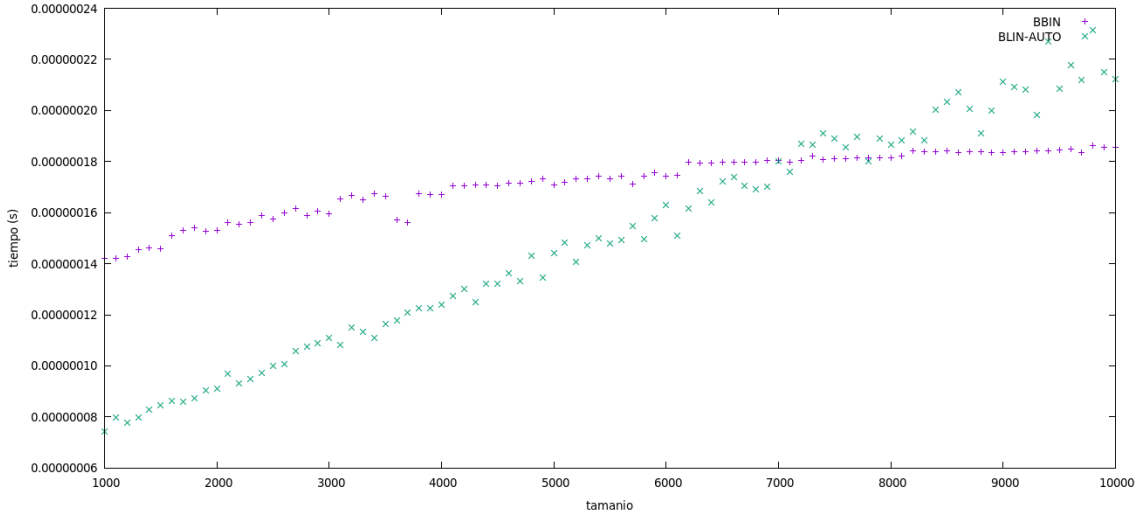
Efectivamente, en cuanto a OBs, BLin_auto está empezando a funcionar, cuando antes teníamos algún caso de unas 5000 OBs medias, ahora la máxima está por debajo de las 700. Y ahora BBin, que sigue siendo una recta, la estamos viendo más de cerca y realmente no estaba en 0, aunque sus OBs medias siguen siendo muy bajas.



Los tiempos empiezan a descender en el caso de tiempos medios. BLin_auto empieza a ser mucho más uniforme, creciente como siempre, pero con tiempos mucho más bajos que con anterioridad. En BBin empezamos a apreciar fluctuaciones en esa “recta”, y aunque para tamaños bajos, empiezan a ser bastante parejos, para grandes tamaños, BBin sigue aplastando en ranking a las búsquedas lineales.



Finalmente, en el último caso (n_veces = 10000), BLin_auto encuentra su posición como mejor opción de búsqueda para pequeños tamaños. Las OBs medias para ambos son muy bajas, pero el descenso de BLin_auto es muy llamativo, su característica propia empieza a ser muy efectiva. Ahora que la escala de OBs medias ha bajado lo suficiente podemos observar la forma logarítmica binaria de BBin, ya ha dejado de parecer una recta en el fondo de la gráfica, y aunque las dos gráficas sean crecientes y bastante regulares, BBin sigue mandando en operaciones básicas para tamaños medios y grandes.

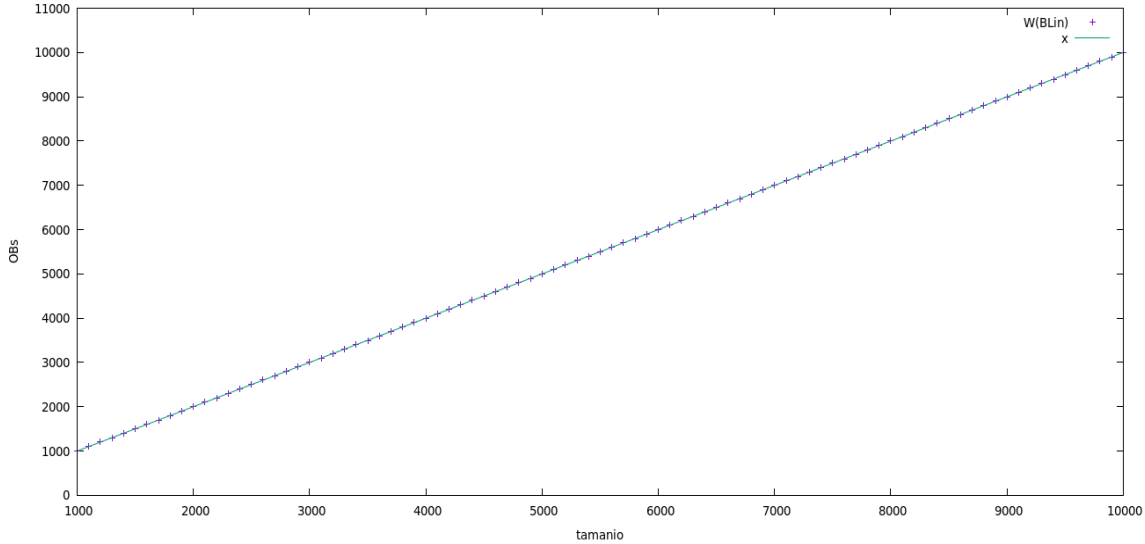


Como pasaba con las operaciones básicas medias, los tiempos medios siguen la misma dinámica. Para BLin_auto, los tiempos han bajado tanto en cómputos globales que se puede apreciar que BBin realmente no era lineal, sino que reluce su forma logarítmica. Como pasaba con las OBs, se comprueba a simple vista que BLin_auto es más eficiente para tamaños pequeños del diccionario, mientras que BBin, maneja mucho mejor los tamaños medios y grandes, a pesar de que, lógicamente, las dos funciones crecen con el tamaño.

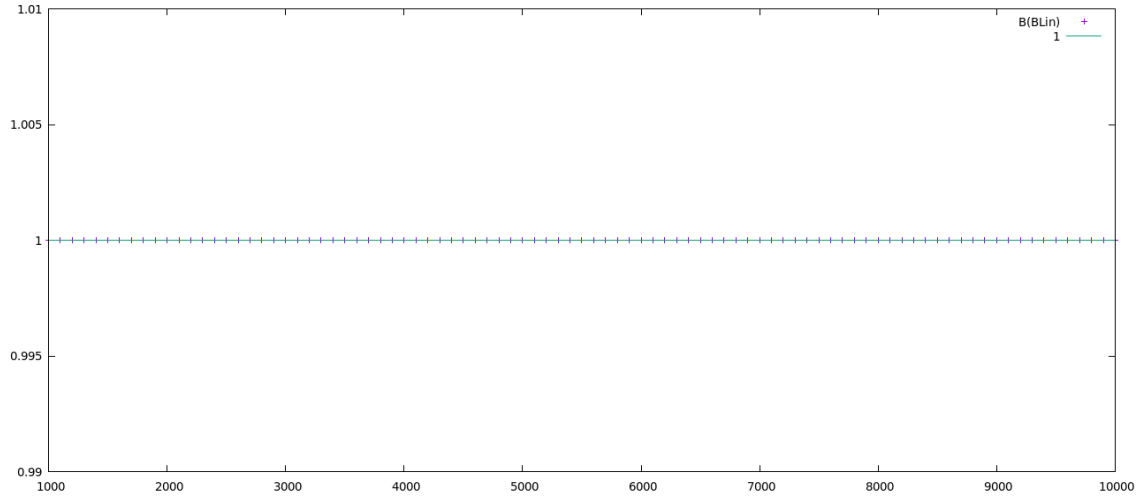
4. Cuestiones teóricas

1. La operación básica de los tres algoritmos de búsqueda es la comparación de claves.

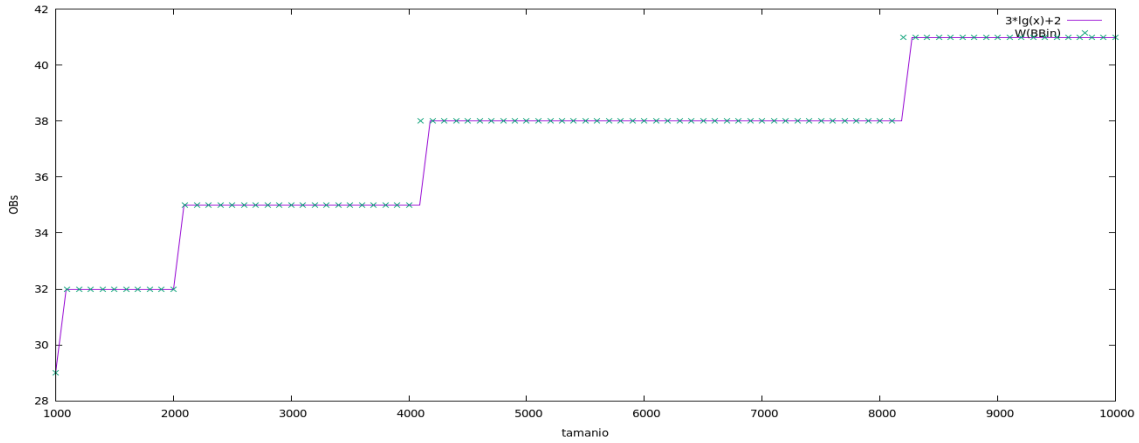
2. El caso peor de BLin es tener que recorrer toda la lista para encontrar la clave, es decir O(N), pero exactamente N.



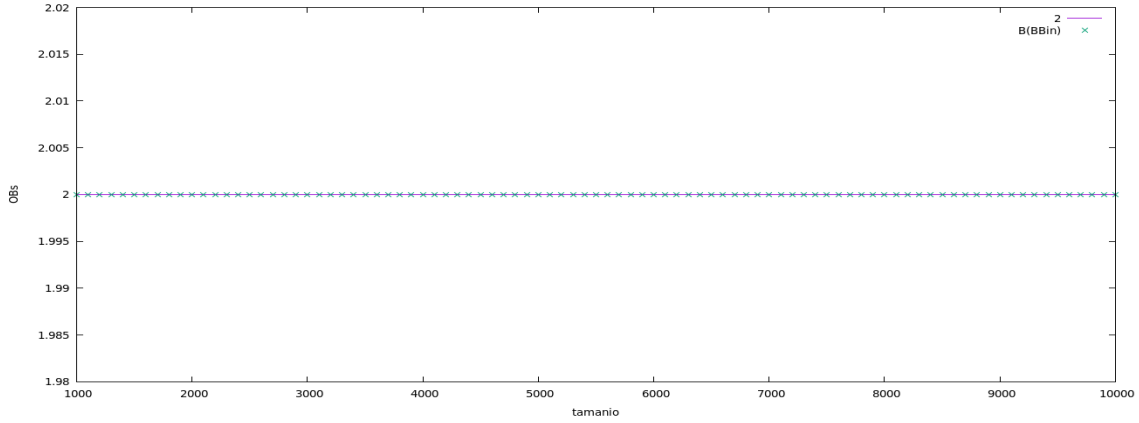
El caso mejor, evidentemente, es encontrar la clave en la primera posición de la tabla y hacer una sola CDC.



Para BBin, el caso peor es tener que llegar hasta la hoja más profunda de su "árbol de búsqueda". Por esto, el caso peor tiene un coste de O(lg(N)), más concretamente el techo de lg(N).

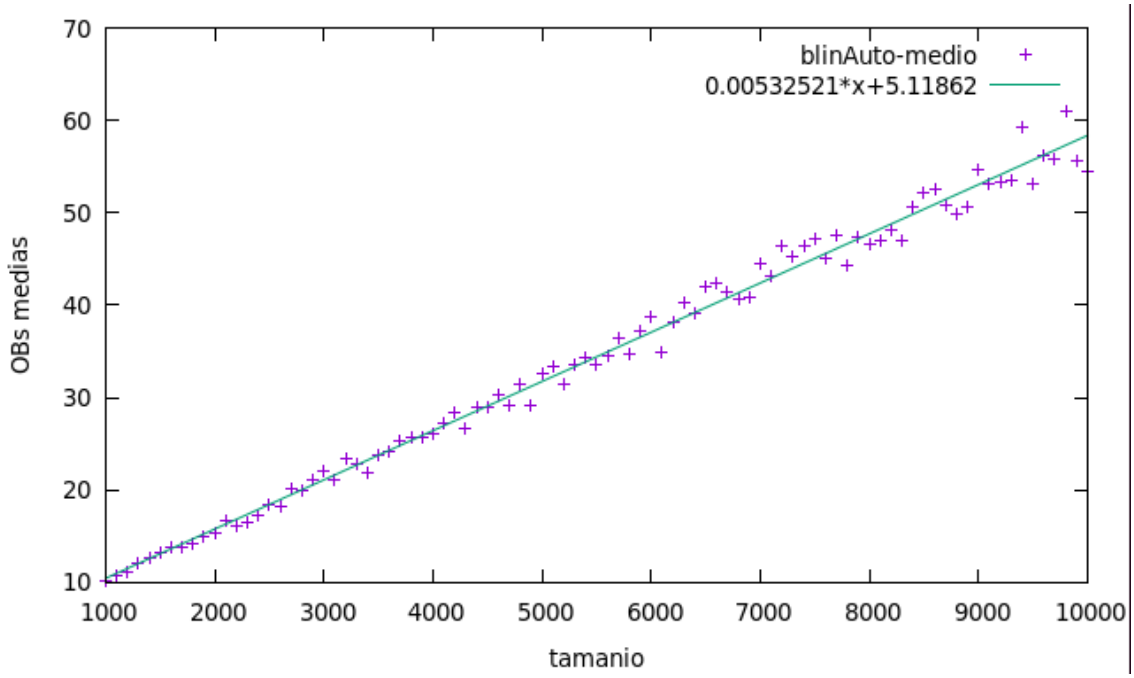


Su caso mejor, como pasaba antes, es encontrar la clave en el nodo raíz del "árbol de búsqueda" o, mejor dicho, en la posición media de la tabla. Por esto, su coste es de O(1), sin embargo, como la implementación es recursiva y primero se comprueba si la posición primera y última son correctas, le tenemos que sumar una comparación.



3. La posición de las claves en la tabla cuando se usa BLin_auto varía de tal forma que, las claves más buscadas y con mayor probabilidad de ser buscadas de nuevo, se vayan situando en las posiciones más próximas al inicio de la tabla.

4.



Como muestra la gráfica, el caso medio de OBs para blin_auto con tamaños de 10000 y con n_veces = 10000, sigue teniendo ese aspecto lineal típico de los costes de O(N), aun que como se puede ver en el ajuste lineal podríamos hablar de costes que se acercan a N/200 o lo que es lo mismo 0.01*N/2, así que podemos hablar de órdenes de O(N/2).

5. El algoritmo de búsqueda binaria, es muy eficaz por el hecho de que la tabla de números está ordenada, es decir, al hacer la comparación de claves, si sale mayor o menor que la primera clave comparada (la posición media de la tabla), sabe que tiene que buscar hacia cierta zona de la tabla, quitándose una gran cantidad de números a comparar (la mitad de cada subtabla) porque todos serán menores o mayores que el comparado y, por tanto, quitarse una gran cantidad de comparaciones y pérdidas de tiempo.

De esta manera, con muy pocas comparaciones (log(N)) es capaz de encontrar o no encontrar la clave en búsqueda.

5. Conclusiones finales.

Como conclusión, deducimos que la búsqueda binaria es increíblemente efectiva para tablas ordenadas cuando se buscan claves pocas veces, sin embargo, cuando se buscan las claves repetidas veces y de forma dispar, funciona muy bien para tamaños medios y grandes de la tabla.

BLin, en general, es muy costoso, salvo para cuando buscamos claves que están en las primeras posiciones de la tabla. Es por esto que la implementación de la característica principal de BLin_auto optimiza tanto a este algoritmo, ya que hace que las claves más buscadas, y que es muy probable que se vuelvan a buscar, pasen a las primeras posiciones y, por tanto, reluzca más el algoritmo lineal.