Análisis de Algoritmos 2022/2023

Práctica 3

Vuestros nombres, Grupo.

Daniel Cruz, Daniel Birsan.

1. Introducción.

Vamos a utilizar un TAD diccionario, en el cual tendremos un array, ordenado o desordenado. Aplicaremos búsquedas lineales y binarias con el objetivo de analizar sus complejidades y diferencias entre ambas.

2. Objetivos

Aquí indicáis el trabajo que vais a realizar en cada apartado.

2.1 Apartado 1

Objetivos del apartado 1.

Se trata de crear el TAD diccionario con sus respectivas primitivas para iniciarlo, insertar, poblarlo y liberarlo. Además implementaremos los algoritmos de búsqueda lineal, binaria y lineal autoorganizada.

2.2 Apartado 2

Con ayuda del TAD diccionario, vamos a evaluar el rendimiento de los distintos algoritmos de búsqueda implementados en el apartado anterior, implementando varias funciones en times.c con este objetivo.

3. Herramientas y metodología

Aquí ponéis qué entorno de desarrollo (Windows, Linux, MacOS) y herramientas habéis utilizado (Netbeans, Eclipse, gcc, Valgrind, Gnuplot, Sort, uniq, etc) y qué metodologías de desarrollo y soluciones al problema planteado habéis empleado en cada apartado. Así como las pruebas que habéis realizado a los programas desarrollados.

Ambos hemos utilizado Ubuntu para el desarrollo del proyecto, Visual Studio Code como editor de texto, gcc para la compilación, Valgrind para detectar posibles fugas de memoria o leaks, gnuplot para la creación de las gráficas. Hemos utilizado git y github para el control de versiones a lo largo del proyecto.

3.1 Apartado 1

Hemos utilizado nuestro editor de código para modificar todos los ficheros .c y .h e implementar las funciones pedidas de diccionarios, además de los algoritmos de búsqueda pedidos, hemos hecho se correspondiente commit en github y continuado con el trabajo, con valgrind nos hemos asegurado de su correcto funcionamiento, compilando con gcc.

3.2 Apartado 2

Hemos implementado las funciones de times.c para utilizarlas posteriormente en las fabricación de las gráficas con gnuplot. Hemos compilado estas funciones con gcc y revisado con Valgrind. Finalmente para seguir con nuestro control de versiones hemos hecho otro commit en nuestro repositorio remoto.

4. Código fuente

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

4.1 Apartado 1

```c

#include "search.h"

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

/**
 *  Key generation functions
 *
 *  Description: Receives the number of keys to generate in the n_keys
 *               parameter. The generated keys go from 1 to max. The
 *         keys are returned in the keys parameter which must be
 *         allocated externally to the function.
 */

/**
 *  Function: uniform_key_generator
 *               This function generates all keys from 1 to max in a sequential
 *               manner. If n_keys == max, each key will just be generated once.
 */
void uniform_key_generator(int *keys, int n_keys, int max)
{
  int i;

  for(i = 0; i < n_keys; i++) keys[i] = 1 + (i % max);

  return;
}

/**
 *  Function: potential_key_generator
 *               This function generates keys following an approximately
 *               potential distribution. The smaller values are much more
 *               likely than the bigger ones. Value 1 has a 50%
 *               probability, value 2 a 17%, value 3 the 9%, etc.
 */
void potential_key_generator(int *keys, int n_keys, int max)
{
  int i;

  for(i = 0; i < n_keys; i++)
  {
    keys[i] = .5+max/(1 + max*((double)rand()/(RAND_MAX)));
  }

  return;
}

PDICT init_dictionary (int size, char order)
{
  PDICT dic;

  dic = (PDICT) malloc(sizeof(DICT));
  if (!dic)
    return NULL;

  dic->table = (int *) malloc(sizeof(int)*size);
  if (!dic->table) {
    free(dic);
    return NULL;
  }
  dic->size = size;
  dic->n_data = 0;
  dic->order = order;
  return dic;
}

```

```c
void free_dictionary(PDICT pdict)
{
  if (!pdict)
    return;

  if (pdict->table != NULL)
    free(pdict->table);
  free(pdict);
}

int insert_dictionary(PDICT pdict, int key)
{
  int a, j;

  if (!pdict)
    return ERR;

  if (pdict->n_data == pdict->size) {
    if (realloc(pdict->table, sizeof(pdict->table)*2 + 100) == NULL) {
      free(pdict->table);
      return ERR;
    }
    pdict->size *= 2;
    pdict->size += 100;
  }
  a = 0; j = 0;
  pdict->table[++pdict->n_data - 1] = key;
  if (pdict->order == SORTED) {
    a = pdict->table[pdict->n_data-1];
    j = pdict->n_data-2;
    while (j >= 0 && pdict->table[j] > a) {
      pdict->table[j+1] = pdict->table[j];
      j--;
    }
    pdict->table[j+1] = a;
  }
  return OK;
}

int massive_insertion_dictionary (PDICT pdict,int *keys, int n_keys)
{
  int i;

  if (!pdict || !keys || n_keys <= 0)
    return ERR;

  i = 0;
  while (i < n_keys) {
    if (insert_dictionary(pdict, keys[i]) == ERR)
      return ERR;
    i++;
  }
  return OK;
}

int search_dictionary(PDICT pdict, int key, int *ppos, pfunc_search method)
{
  return method(pdict->table, 0, pdict->size - 1, key, ppos);
}
```

```c
int bin_search(int *table,int F,int L,int key, int *ppos)
{
  int m = 0;
  int c = 0;

  if (!table || F > L || !ppos)
    return ERR;

  while(F <= L)
  {
    c++;
    m = (F+L)/2;
    if (table[m] == key)
    {
      (*ppos) = m;
      return c;
    }
    else if (key < table[m])
    {
      L = m--;
    }
    else
    {
      F = m++;
    }
  }

  (*ppos) = NOT_FOUND;
  return NOT_FOUND;
}

int lin_search(int *table,int F,int L,int key, int *ppos)
{
  int i;
  int j = 0;

  if (!table || !ppos)
    return ERR;

  i = 0;
  for (i = F; i <= L; i++) {
    if (table[i] == key) {
      *ppos = i;
      return i+1;
    }
  }
  *ppos = NOT_FOUND;
  while (j <= L) {
    printf("%d ", table[j++]);
  }
  printf("\n\n");
  return i;
}

int lin_auto_search(int *table,int F,int L,int key, int *ppos)
{
  int aux = 0;
  int c = 0;

  if (!table || F > L || !ppos) return ERR;

  if(table[0] == key)
  {
    (*ppos) = 0;
    return ++c;
  }

  F++;
  while (F <= L)
  {
    c++;
    if (table[F] == key)
    {
      aux = table[F];
      table[F] = table[F-1];
      table[F-1] = aux;
      (*ppos) = F;
      return ++c;
    }
    F++;
  }
  return ERR;
}
```

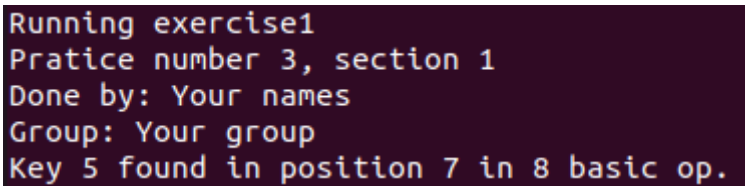
4.2 Apartado 2

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "times.h"
#include "sorting.h"
#include "search.h"
#include "permutations.h"


short average_search_time(pfunc_search metodo, pfunc_key_generator generator, int order, int N, int n_times, PTIME_AA ptime)
{
  PDICT dic;
  int *perm, *keys, i, pos = 0, ob = 0;
  int maxob = -(__INT_MAX__), minob = __INT_MAX__;
  clock_t start, end;
  float avg = 0, avgob = 0;

  dic = init_dictionary(N, order);
  if (!dic)
    return ERR;

  perm = generate_perm(N);
  if (!perm) {
    free_dictionary(dic);
    return ERR;
  }

  if (massive_insertion_dictionary(dic, perm, N) == ERR) {
    free_dictionary(dic);
    free(perm);
    return ERR;
  }

  keys = (int *) malloc(sizeof(int)*(N*n_times));
  if (!keys) {
    free_dictionary(dic);
    free(perm);
    return ERR;
  }

  uniform_key_generator(keys, N*n_times, N*n_times);
  ptime->N = N;
  ptime->n_elems = N*n_times;
  for (i = 0; i < ptime->n_elems; i++) {
    start = clock();
    if ((ob = metodo(dic->table, 0, N, keys[i], &pos)) == ERR) {
      free_dictionary(dic);
      free(perm);
      free(keys);
      return ERR;
    }
    if (ob <= minob)
      minob = ob;
    if (ob >= maxob)
      maxob = ob;
    end = clock();
    avgob += ob;
    avg += (float) start / end;
  }
  ptime->time = (float) avg / ptime->n_elems;
  ptime->average_ob = (float) avgob / ptime->n_elems;
  ptime->max_ob = maxob;
  ptime->min_ob = minob;
  free_dictionary(dic);
  free(perm);
  free(keys);
  return OK;
}


short generate_search_times(pfunc_search method, pfunc_key_generator generator, int order, char* file, int num_min, int num_max, int incr, int n_times)
{
  int i;
  TIME_AA times;
  FILE *f;

  f = fopen(file, "w");
  if (!f)
    return ERR;
  fclose(f);
  for ( i = num_min; i <= num_max; i+=incr)
  {
    if (average_search_time(method, generator, order, i, n_times ,&times) == ERR)
      return ERR;
    if (save_time_table(file, &times, i) == ERR)
      return ERR;
  }
  return OK;
}

short save_time_table(char* file, PTIME_AA ptime, int n_times)
{
  FILE *f;

  f = fopen(file, "a+");
  if (!f)
    return ERR;
  fprintf(f, "%d %f\n", n_times, ptime->average_ob);
  fclose(f);
  return OK;
}

```

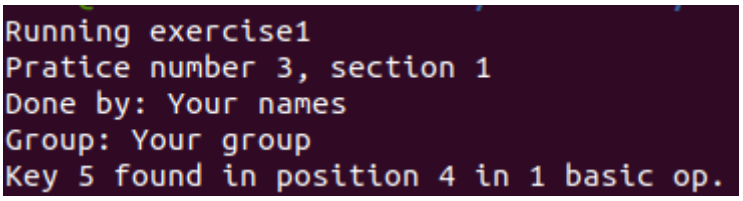
5. Resultados, Gráficas

5.1 Apartado 1

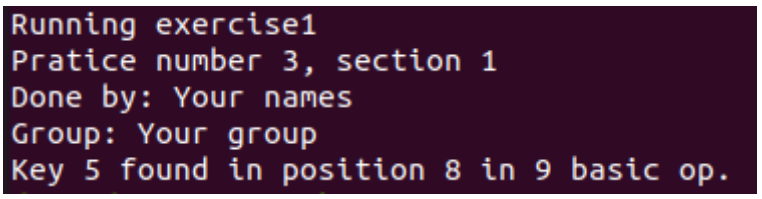
Para lin_sort elegiremos el tamaño de la permutación, la clave a buscar y que la permutación no esté ordenada:



Para bin_sort elegiremos el tamaño, la clave y que la tabla esté ordenada porque sino no funciona el algoritmo:

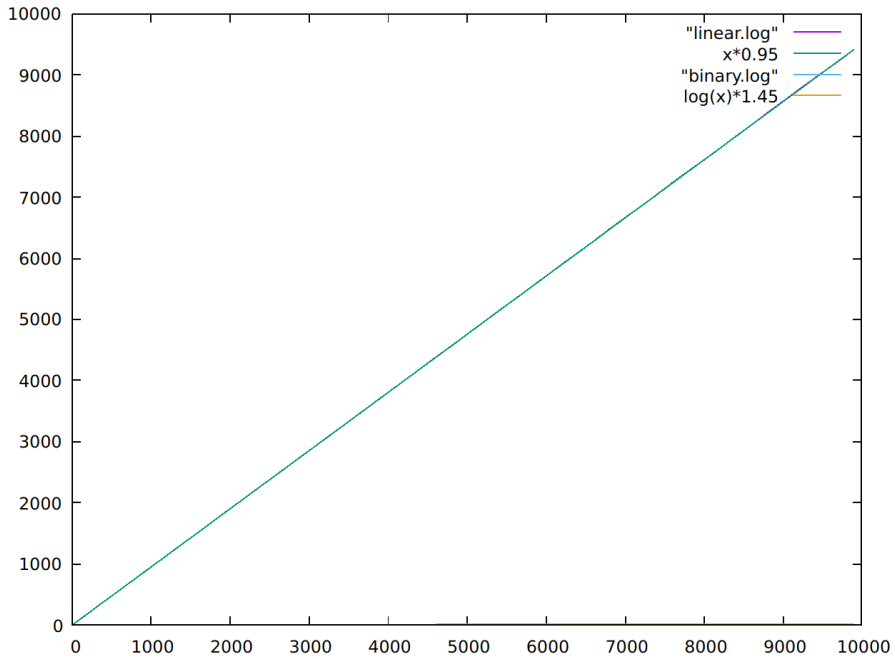


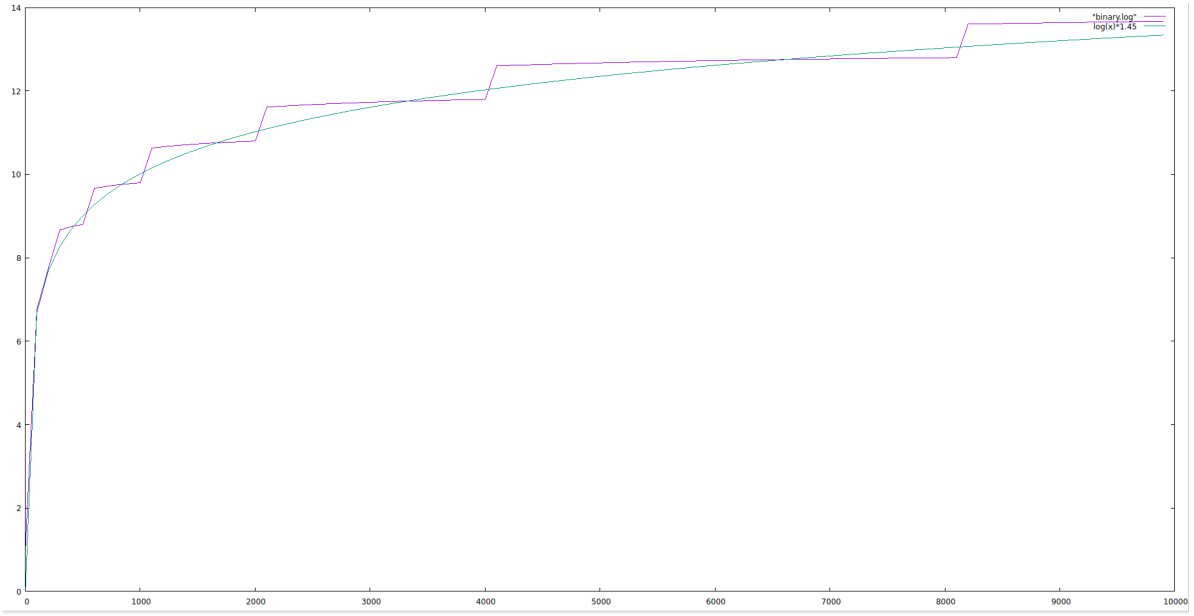
Para lin_auto_search elegiremos el tamaño, la clave y que la tabla esté desordenada:



5.2 Apartado 2

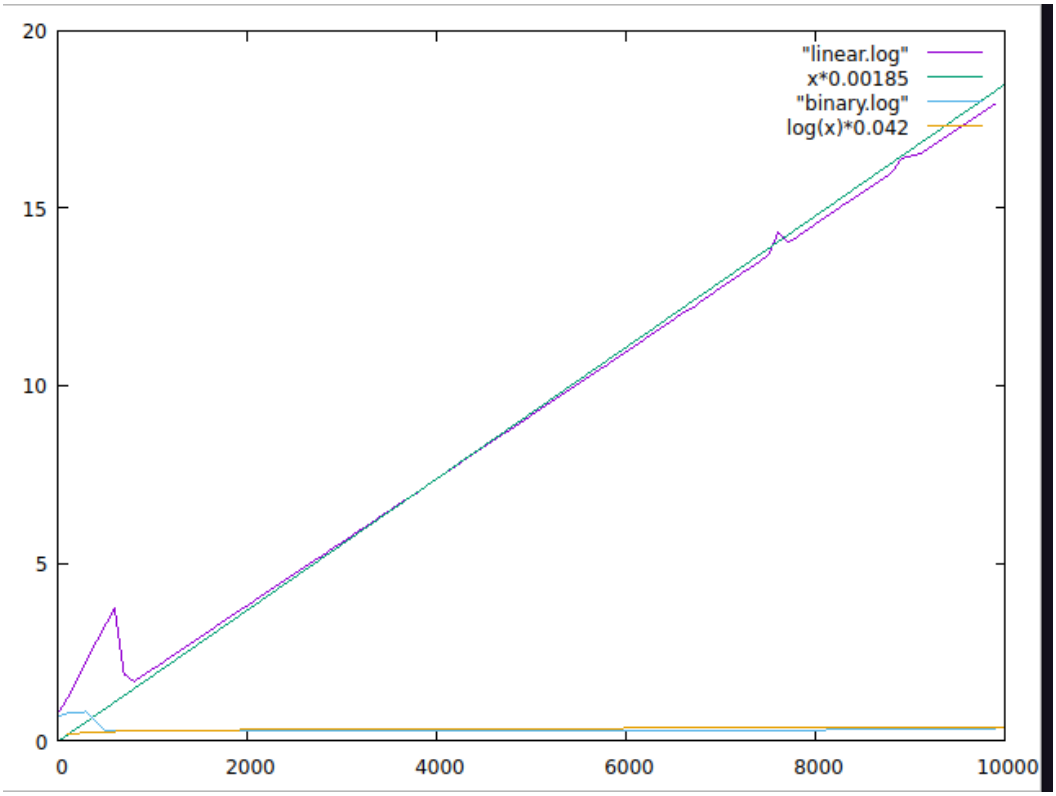
Gráfica comparando el número promedio de OBs entre la búsqueda lineal y la búsqueda binaria, comentarios a la gráfica.





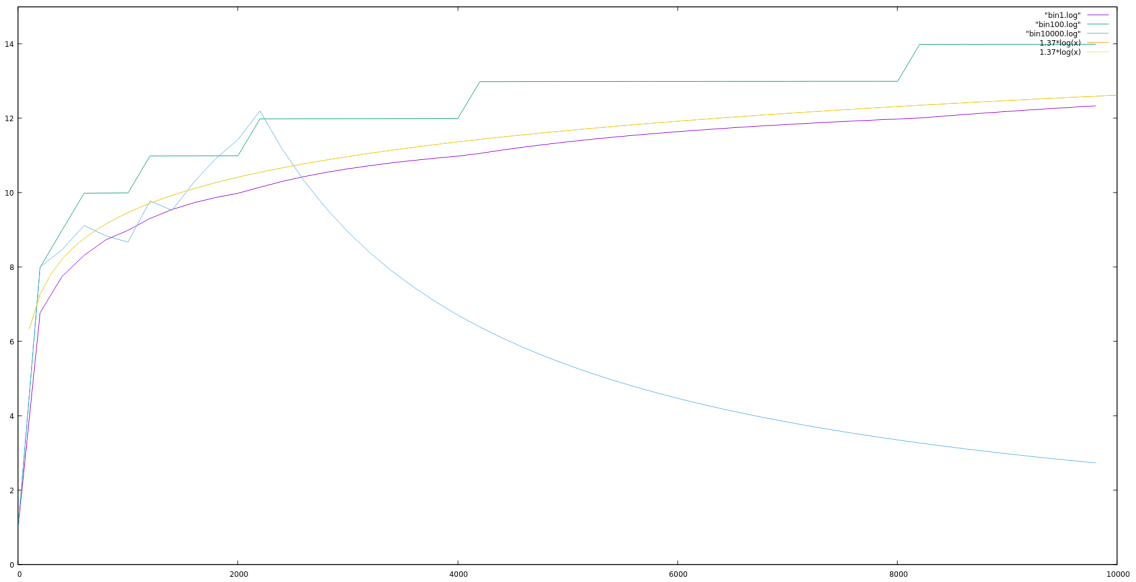
Como se aprecia, la búsqueda lineal es O(n), mucho más grande que O(logn) de la búsqueda binaria, debido a ello, no apreciamos la curva de la búsqueda binaria. en la primera gráfica y hemos puesto una segunda con solo la búsqueda binaria para analizarla mejor. Se ve como la búsqueda binaria efectivamente utiliza O(logn) obs, la curva sale escalonadada dado que los valores de las OB siguen siendo bastante bajos aún con 10000 objetos.

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

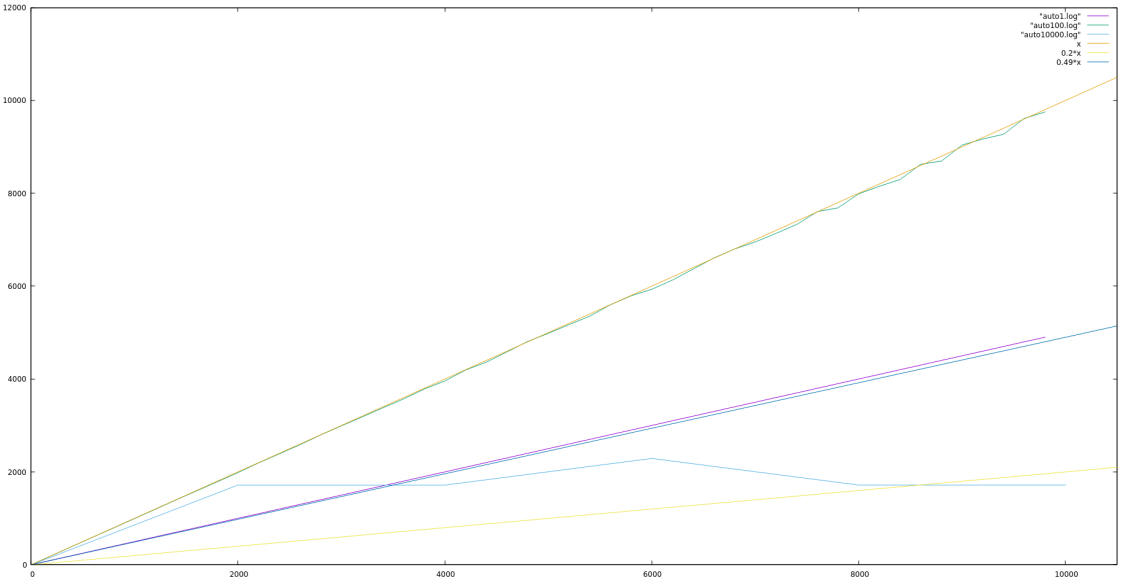


Al igual que en las OB, apreciamos O(n) en la búsqueda lineal y O(logn) en la búsqueda binaria, bastante más lenta la primera como se esperaba. Se observa un pico al principio en ambas búsquedas, probablemente debido a la caché que después de las primeras veces de ejecución guarda datos y el tiempo de ejecución baja(cosas ajenas al programa, del sistema operativo).

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

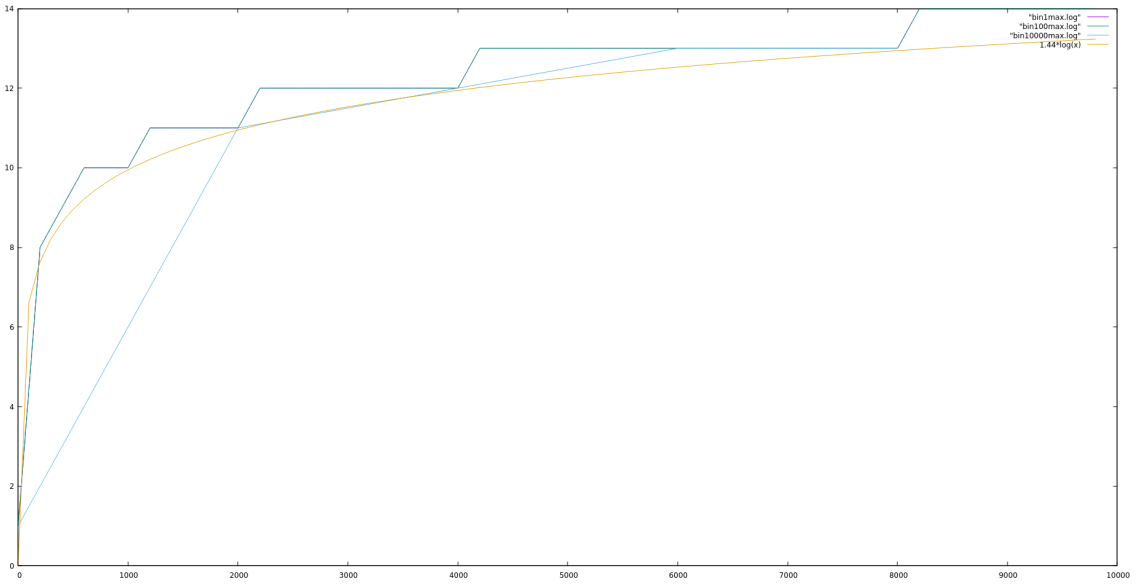


Búsqueda binaria y valor teórico ajustado (1.37*logn) en amarillo, la línea morada se corresponde al n_times 1, la verde a n_times 100 y la azul a n_times 10000.

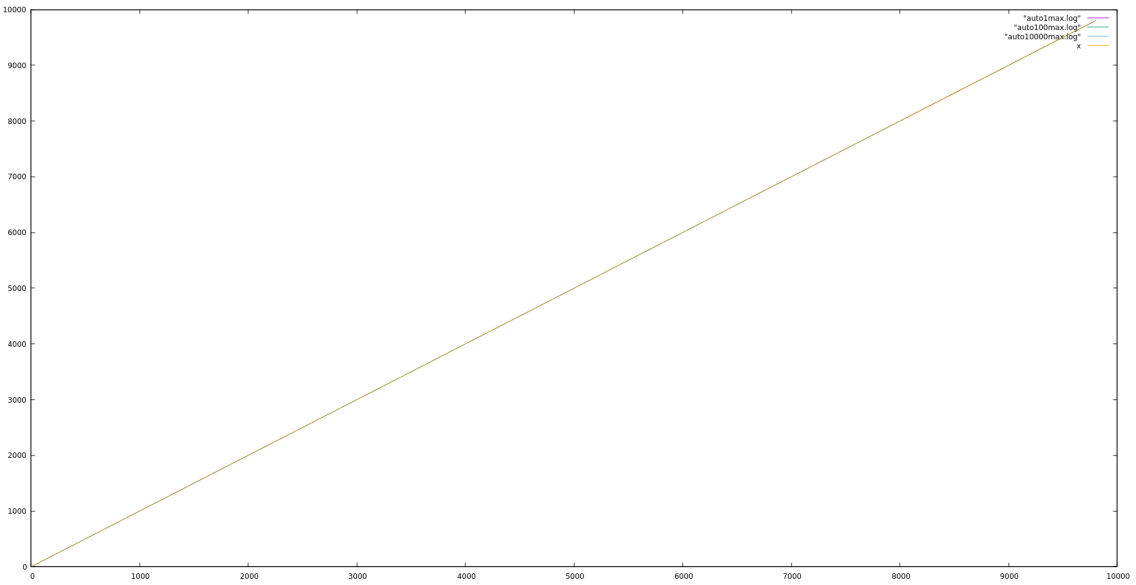


Búsqueda lineal auto organizada y valores teóricos ajustados a cada n_times, según la altura, se representan las obs medias con n_times 1, 100, 10000, respectivamente, el ajuste se ha hecho con n, 0.5n, y 0.2n en los caso respectivos.

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

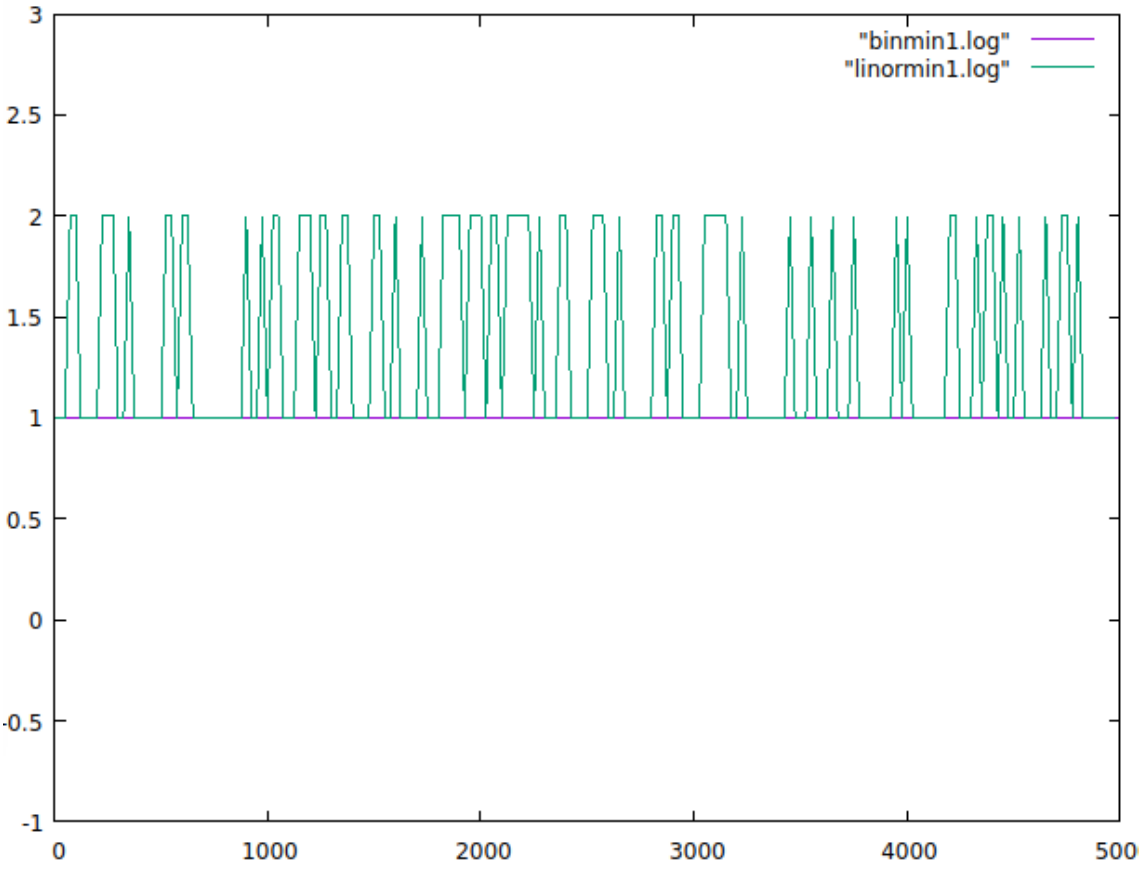


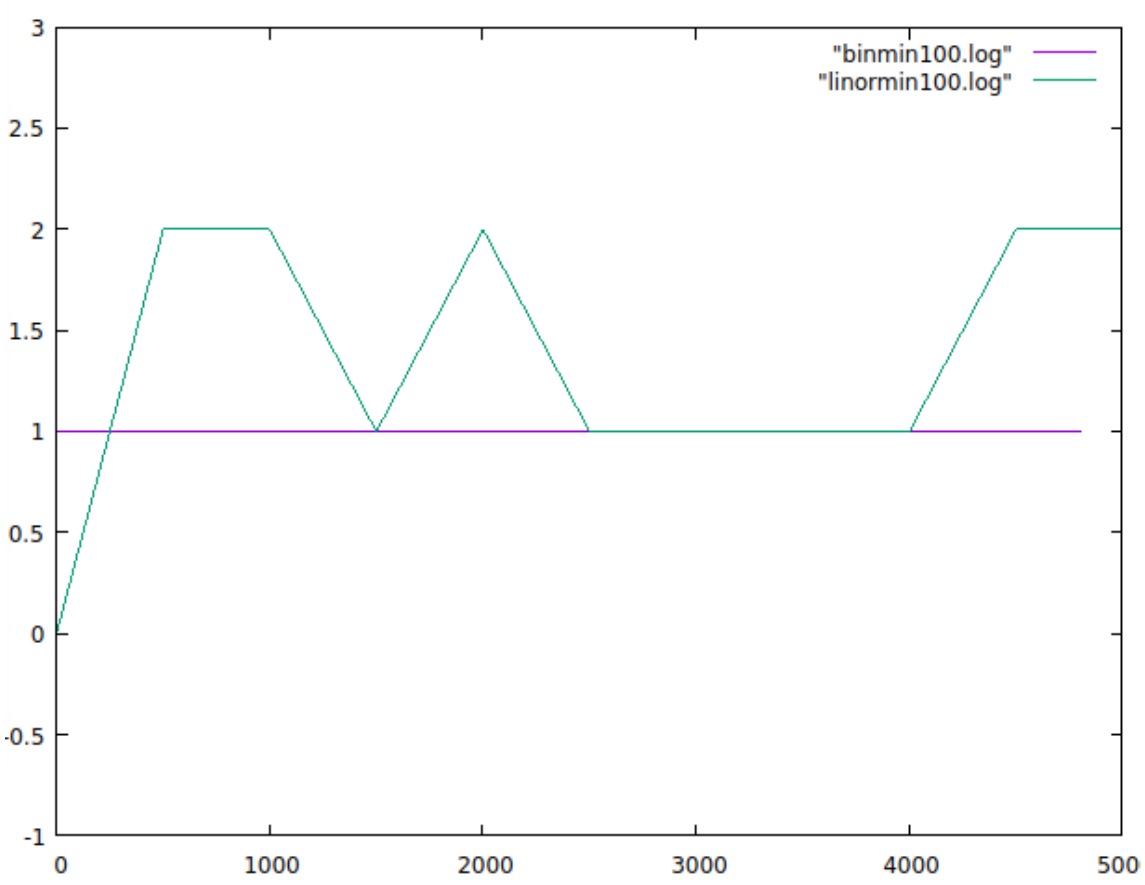
Búsqueda binaria comparando las operaciones básicas máximas, en este caso no hay mucha variación y hemos podido ajustar el valor teórico logn multiplicándolo por 1.44, con n_times 1 y n_times 100 los resultados son muy parecidos (2 de arriba), por otro lado, con n times 10000 (línea más baja), las obs varían un poco más, en sentido positivo.

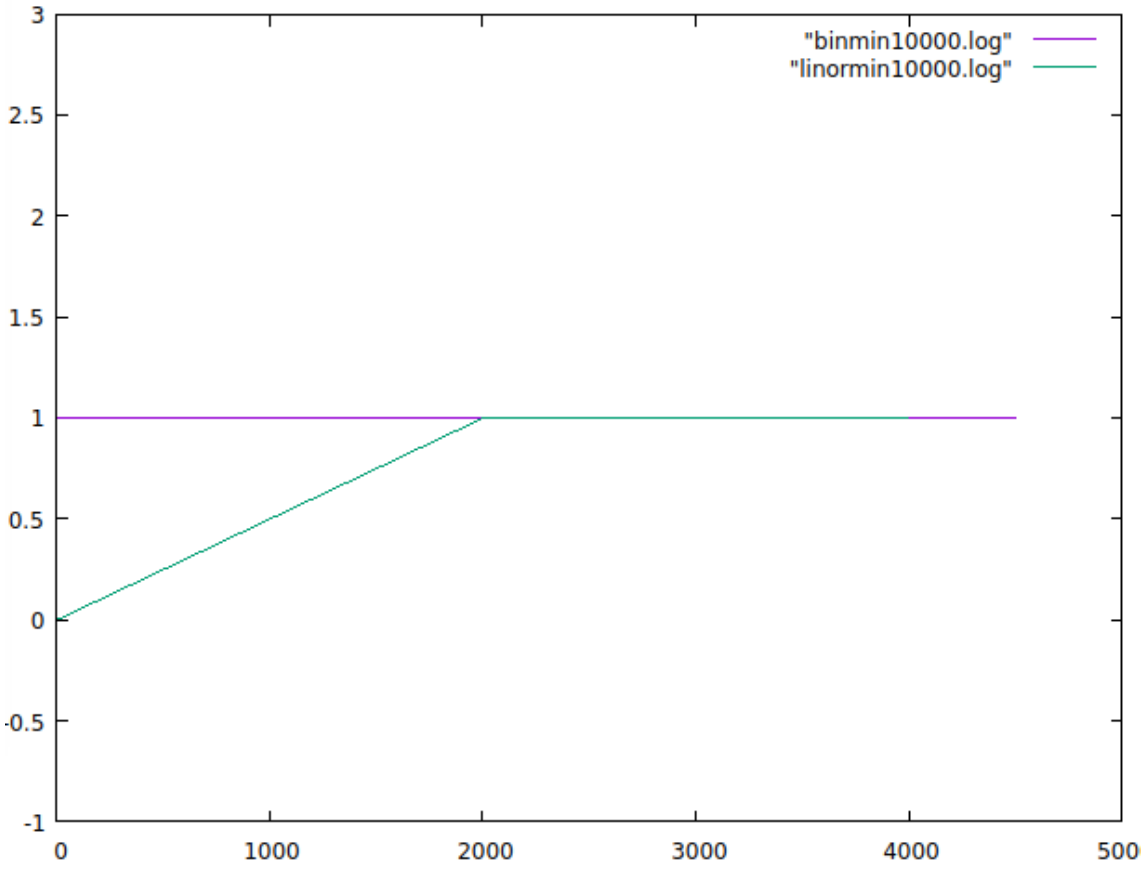


Búsqueda lineal auto organizada comparando el número de obs máximo, como era de esperar los valores son los mismos o prácticamente iguales, lo asemejamos con el valor teórico n.

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

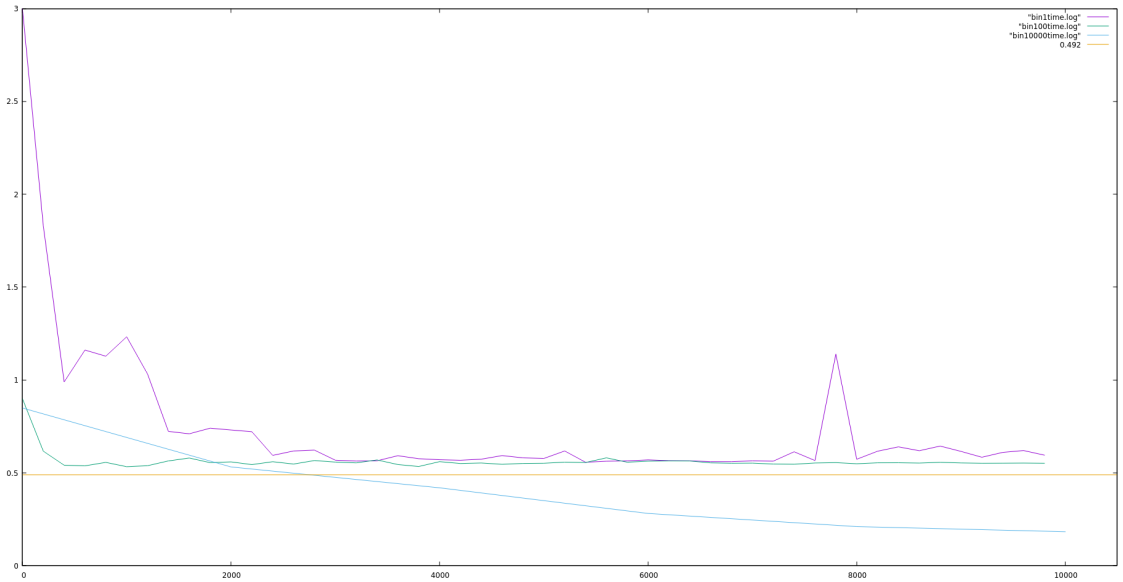



"binmin100.log"
"linormin100.log"
3
2.5
2
1.5
1
0.5
0
-0.5
-1
0
1000
2000
3000
4000
500

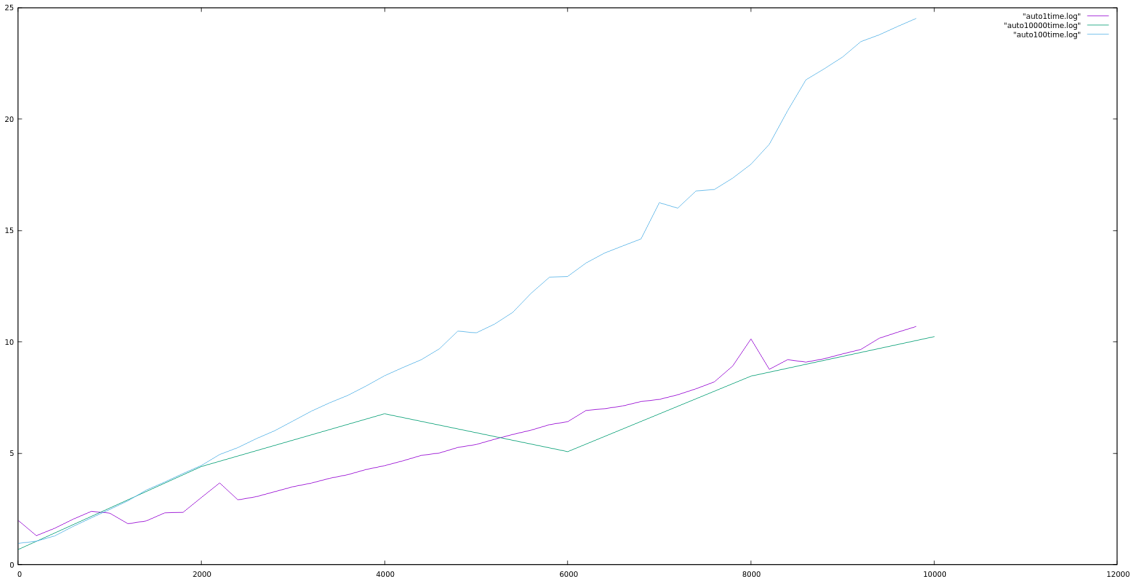

"binmin10000.log"
"linormin10000.log"
3
2.5
2
1.5
1
0.5
0
-0.5
-1
0
1000
2000
3000
4000
500

Como vemos en las gráficas, la búsqueda binaria se mantiene en 1 ob siempre. La lineal autoorganizada varía entre 1 y 2 con 1 y 100 n_times, y es 1 con 10000 n_times.

Gráfica comparando el tiempo medio de reloj entre la búsqueda binaria y la búsqueda lineal auto organizada (para los valores de n_times=1, 100 y 10000), comentarios a la gráfica.



Búsqueda binaria comparando el tiempo medio de ejecución, podemos ver que este tipo de búsqueda es más eficiente cuantas más veces se ejecuta, para asemejar el valor teórico calculamos la media de los tiempos de ejecución, en este caso 0,492 y lo representamos en la gráfica.



Búsqueda lineal auto organizada comparando los tiempos medios de ejecución, sabemos que la búsqueda auto organizada es útil cuantas más veces se ejecute sobre una misma clave, podemos observar que para n time 10000 se consigue el

menor tiempo de ejecución, sin embargo, el n times 100 hace que se llegue al tiempo de ejecución máximo por la dispersión en la búsqueda de claves al ejecutarse tan pocas veces.

5. Respuesta a las preguntas teóricas.

5.1 Pregunta 1

En ambas búsquedas lineales y en la binaria la operación básica es la comparación de clave del elemento que buscamos y la posición de la tabla en la que estamos.

5.2 Pregunta 2

Los tiempos para el peor caso son O(n) en búsqueda lineal y O(logn) en búsqueda binaria.

5.3 Pregunta 3

En este tipo de búsqueda hacemos un swap de elemento buscado con el elemento anterior al elemento buscado. Por tanto si ejecutamos 100 veces esta búsqueda sobre una tabla de 10 elementos, los elementos más buscados estarán en las posiciones primeras mientras qué los menos buscados se encontrarán en las últimas posiciones del array. Esta búsqueda es útil para reducir el coste cuando se buscan varias veces los mismos elementos.

5.4 Pregunta 4

Considerando que el 80% de nuestras consultas buscan el 20% de las claves por la regla 80/20 el coste medio sería N/10 ya que N/2 * ⅕, proveniente del 20/100, da N/10.

5.5 Pregunta 5

El algoritmo bin_search() de búsqueda binaria para tablas ordenadas es efectivo dado que consigue tener una complejidad de O(logn), bastante mejor que O(n) (búsqueda lineal). Esto es debido a que al ir descartando los elementos mayores o menores que el elemento que estamos buscando nos ahorramos muchas todas esas comparaciones descartadas

6. Conclusiones finales.

Hemos aprendido a implementar los algoritmos de búsqueda más conocidos y aplicándolos a ejemplos prácticos hemos obtenido resultados que nos indican en qué situaciones es más óptimo utilizar uno u otro, además hemos fortalecido la búsqueda de claves en diccionarios que puede ser muy relevante en un futuro.

Como conclusión hemos obtenido que la búsqueda lineal es poco eficiente a no ser que la permutación tenga pocos elementos, sin embargo la binaria es muy eficaz pero necesita una permutación ordenada. Por otro lado, la búsqueda lineal auto organizada solo expondrá su eficiencia cuando se ejecute numerosas veces sobre las mismas claves, si solo se ejecuta una vez, tiene el mismo coste que la lineal.