Análisis de Algoritmos 2022/2023

Práctica 3

Andrés Goldwasser, Ignacio Nuñez, Grupo 1271.

Código	Gráficas	Memoria	Total

1. Introducción.

En esta práctica, los objetivos constan de la implementación de un TAD “diccionario”, de funciones de búsquedas en esta estructura, y de un análisis de la eficacia de estas.

2. Objetivos

2.1 Apartado 1

Para este apartado, se busca la implementación de múltiples funciones. Varias permiten la utilización del TAD, mientras que las últimas tres (las denominadas “x_search”) se encargarán de búsqueda de elementos en variables que utilicen esta estructura.

2.2 Apartado 2

En el apartado 2 el objetivo es implementar funciones capaces de medir la eficiencia de las funciones de búsqueda. Realmente solo 1 permite verificar la eficiencia, pero han de ser escritas varias para facilitar el proceso y el análisis de las distintas funciones de búsqueda.

3. Herramientas y metodología

Todo el proceso de la práctica se ha llevado a cabo en el entorno de programación Visual Studio Code.

3.1 Apartado 1

Las funciones init_dictionary y free_dictionary no costaron mucho trabajo, siendo estas bastante estándar en el desarrollo de TADs. Insert_dictionary se realizó como se indica, traduciendo el seudocódigo del enunciado a C para su implementación; luego massive_insertion_dictionary simplemente realiza llamadas a esta función n_key veces. Finalmente, search_dictionary es otra función que llama a otras y devuelve la posición de la clave en el diccionario.

Las funciones de búsqueda se implementaron a modos distintos: para bin_search se tradujo su seudocódigo; lin_search se implementó mediante razonamiento del equipo, y lin_auto_search fue una repetición de lin_search pero sumándole un procedimiento de intercambio de elementos entre el buscado y el que se encuentre antes de este en la lista.

3.2 Apartado 2

Para el apartado 2 se siguieron los ejemplos proporcionados por las funciones realizadas en la práctica 1, modificando estos para que funcionasen con el TAD. Las comparaciones se realizaron con números grandes, y las tablas de datos se proporcionan en el punto #5.

4. Código fuente

4.1 Apartado 1

```c
PDICT init_dictionary (int size, char order)
{
  PDICT dict;

  if(size<=0||(order!=1&&order!=0))
    return NULL;

  dict=(PDICT)malloc(sizeof(DICT));
  dict->size=size;
  dict->n_data=0;
  dict->order=order;
  dict->table=(int*)calloc(size, sizeof(int));
  if(dict->table==NULL)
    return NULL;

  return dict;
}

void free_dictionary(PDICT pdict)
{
/*ERROR CONTROL*/
  if(!pdict){
    return;
  }

  free(pdict->table);
  free(pdict);
  return;
}

int insert_dictionary(PDICT pdict, int key){
  int index;

  /*ERROR CONTROL*/
  if(!pdict){
    return ERR;
  }

  /*ERROR CONTROL*/
  if(pdict->n_data == pdict->size){
    return ERR;
  }

  /*ARRAY ISNT SORTED*/
  if(pdict->order == NOT_SORTED){
    pdict->table[pdict->n_data] = key;
  }
```

```
  /*ARRAY IS SORTED*/
  else if(pdict->order == SORTED){
    index = pdict->n_data-1;
    while(index>=0 && key < pdict->table[index]){
      pdict->table[index+1] = pdict->table[index];
      index--;
    }
    pdict->table[index+1]=key;
  }

  pdict->n_data++;

  return OK;
}

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

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

  for(i=0,flag=OK;i<n_keys&&flag==OK;i++){
    flag=insert_dictionary(pdict,keys[i]);
  }

  if(flag!=OK)
    return ERR;
  else
    return OK;
}

int search_dictionary(PDICT pdict, int key, int *ppos, pfunc_search
method)
{
  int ret;
  if(!pdict||!ppos||!method){
    return ERR;
  }

  ret = method(pdict->table, 0, pdict->n_data-1, key, ppos);

  return ret;
}


/* Search functions of the Dictionary ADT */
int bin_search(int *table,int F,int L,int key, int *ppos)
```

```
{
  int i,count;
  if(!table) return ERR;

  if(L<F){
    *ppos=NOT_FOUND;
    return 0;
  }

  i=(L+F)/2;
  count=1;
  if(table[i]==key){
    *ppos=i;
    return count;
  }
  else if(table[i]>key){
    count+=bin_search(table,F,i-1,key,ppos);
  }
  else if(table[i]<key){
    count+=bin_search(table,i+1,L,key,ppos);
  }

  return count;
}

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

  /*CONTROL ERROR*/
  if(!table||!ppos||L<F){
    return ERR;
  }

  for(i=F;i<L;i++){
    count++;
    if(table[i] == key){
      *ppos = i;
      ret = OK;
      break;
    }
  }

  if(ret==OK){
    return count;
  }

      *ppos = NOT_FOUND;
  return ERR;
```

```
}

int lin_auto_search(int *table,int F,int L,int key, int *ppos)
{
  int i, count=0, ret=ERR, aux;

  /*CONTROL ERROR*/
  if(!table||!ppos||L<F){
    return ERR;
  }

  for(i=F;i<L;i++){
    count++;
    if(table[i] == key){
      if(i==0){
        ret = OK;
        break;
      }
      /*SWAP*/
      aux = table[i-1] ;
      table[i-1] = table[i];
      table[i] = aux;

      *ppos = i-1;
      ret = OK;
      break;
    }
  }

  if(ret==OK){
    return count;
  }
      *ppos = NOT_FOUND;
  return ERR;
}
```

4.2 Apartado 2

```
short average_search_time(pfunc_search method, pfunc_key_generator
generator, char order, int N, int n_times, PTIME_AA ptime){
  PDICT dict;
  int i, *perm, *mem, pos;
  double *array_time, *array_ob;
  clock_t start, stop;

  if(!ptime||n_times<1||N<1) return ERR;

  dict=init_dictionary(N,order);
  if(dict==NULL)
```

```
  return ERR;

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

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

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

array_ob=(double*)malloc((N*n_times)*sizeof(double));
if(!array_ob){
  free(perm);
  free_dictionary(dict);
  free(mem);
  return ERR;
}

array_time=(double*)malloc((N*n_times)*sizeof(double));
if(!array_time){
  free(perm);
  free_dictionary(dict);
  free(mem);
  free(array_ob);
  return ERR;
}

generator(mem,n_times*N,N);

for(i=0 ; i < N*n_times ; i ++){

  start = clock();

  array_ob[i] = method(dict->table, 0, N, mem[i], &pos);

  stop = clock();
  array_time[i] = (double) (stop - start) / CLOCKS_PER_SEC;
}
```

```
  ptime->time = media(array_time, N*n_times);
  ptime->average_ob = media(array_ob, n_times*N);
  ptime->n_elems=N*n_times;
  ptime->N=N;
  ptime->max_ob = array_ob[maxa(array_ob, 0, (n_times*N)-1)];
  ptime->min_ob = array_ob[mina(array_ob, 0, (n_times*N)-1)];

  free(array_ob);
  free(array_time);
  free(mem);
  free(perm);
  free_dictionary(dict);

  return OK;
}

short generate_search_times(pfunc_search method, pfunc_key_genera-
tor generator, int order, char* file, int num_min, int num_max, int
incr, int n_times){
  PTIME_AA time;
  int i, ind;
  short check=OK;
  int mem=0;

  if(!file||n_times<1||num_max<num_min||!method||incr<1) return
ERR;

  for(i=num_min ; i <= num_max; i+=incr, mem++);

  time = (PTIME_AA)malloc((mem)*sizeof(TIME_AA));
  if (!time) return ERR;

  for(i=num_min, ind=0; i <= num_max && check==OK; ind++, i+=incr){
    printf("Voy por %d\n",i);
    check=average_search_time(method, generator, order, i, n_times,
&time[ind]);
    if(check==ERR){
      return ERR;
    }
  }

  if(check==OK)
    check=save_time_table(file, time, mem);

  free(time);

  return check;
}
```

5. Resultados, Gráficas

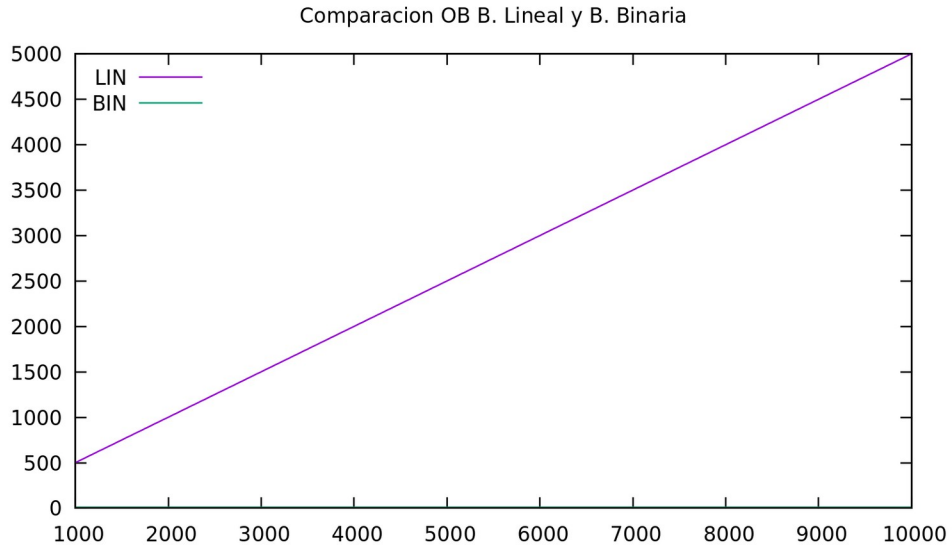
5.1 Apartado 1

Las funciones implementadas para el TAD funcionaron sin fallo, generando 0 errores en valgrind o fugas de memoria. Las funciones de búsqueda todas son capaces de exitosamente encontrar los elementos que se les indique, o indicar que no se encuentra en la lista de ser este el caso.

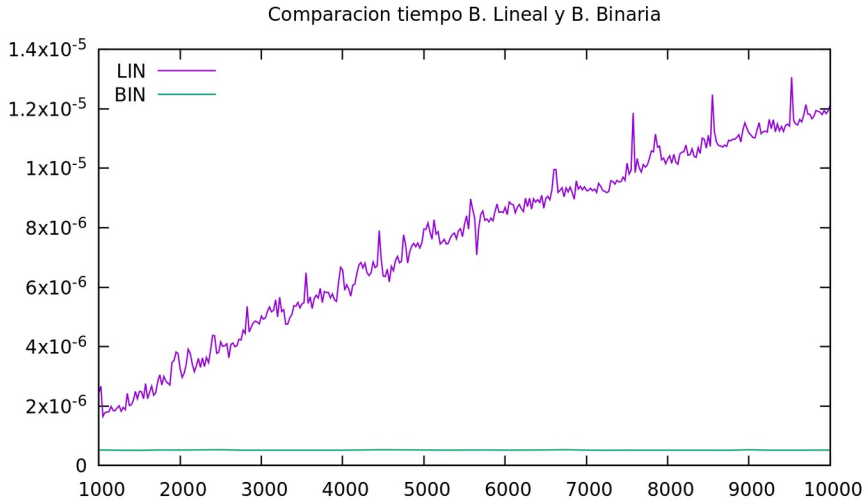
5.2 Apartado 2

Las funciones de medición de tiempo se implementaron exitosamente, y mediante el programa exercise2 pudimos probar las distintas funciones de búsqueda:

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

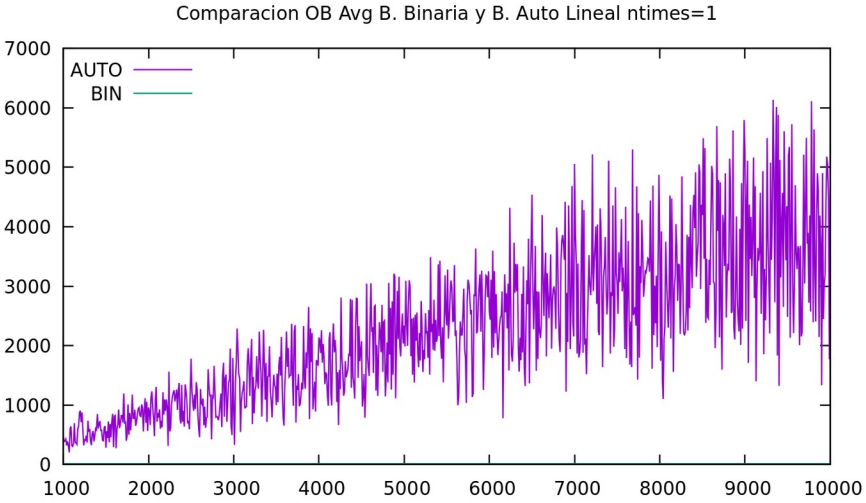


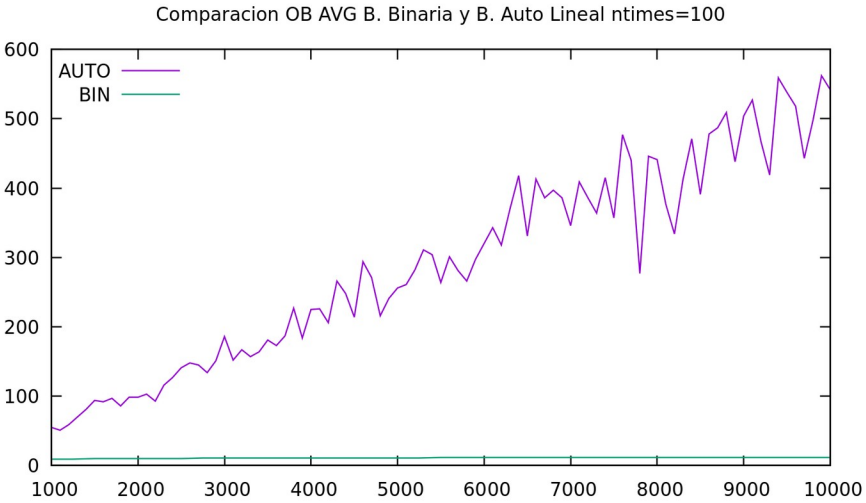
Gráfica comparando el tiempo promedio de reloj entre la búsqueda lineal y la búsqueda binaria, comentarios a la gráfica.

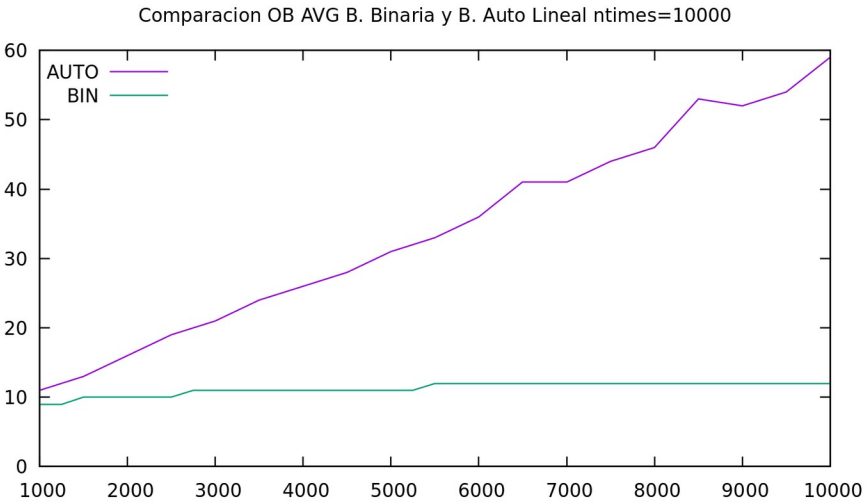


Observamos que la búsqueda binaria es claramente mucho más rápida que la lineal.

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.

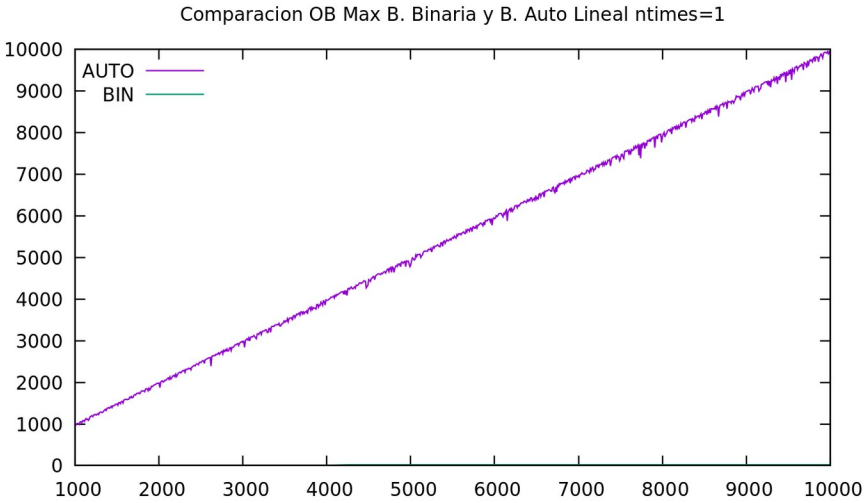


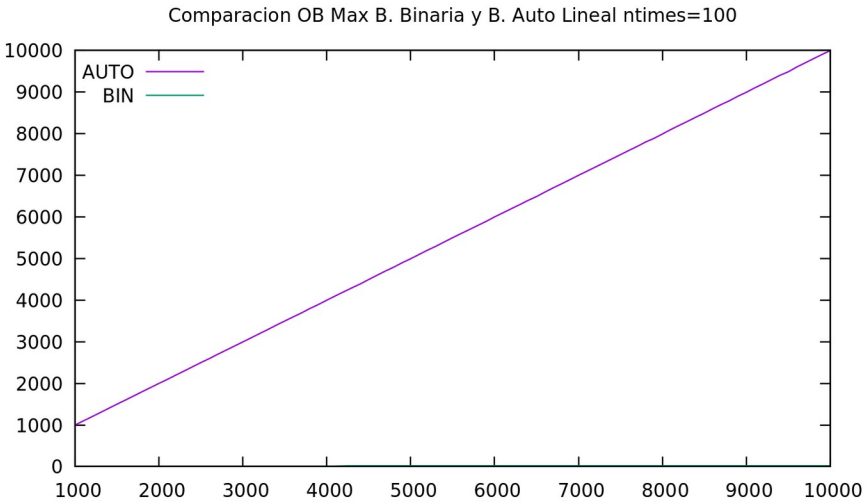


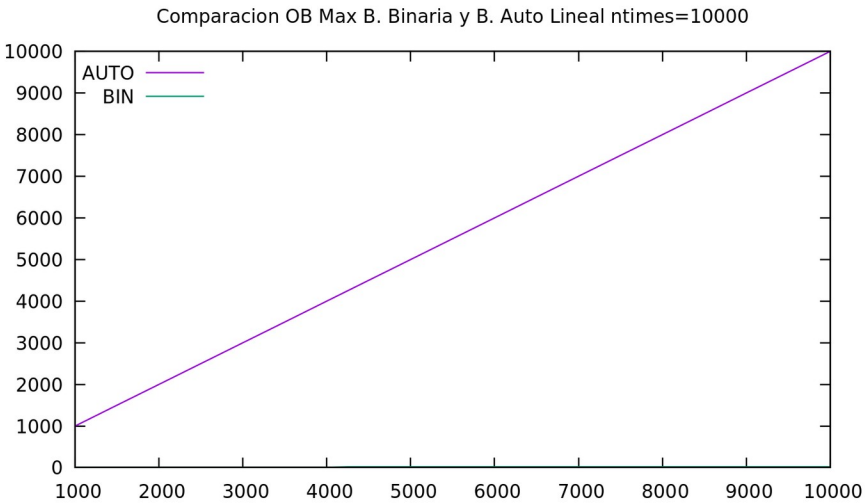


Observamos que para que la búsqueda autolineal sea estable necesita un n_times superior. En caso contrario dependerá en cada caso mucho de la “suerte” de qué potenciales elementos aparezcan, en cuanto el número de n_times aumente todo se termina compensando. Además, también observamos que aunque la búsqueda auto lineal debería tener un orden de O(1) sigue siendo más lento que la búsqueda binaria. Esto es porque para llegar a ese O(1) se tendría que aumentar ntimes mucho más, hasta que tendiera a infinto, algo que sobre la práctica es imposible. Por eso sigue teniendo un comportamiento O(n), con una pendiente muy pequeña, pero que sigue siendo mayor que O(log(n)).

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.

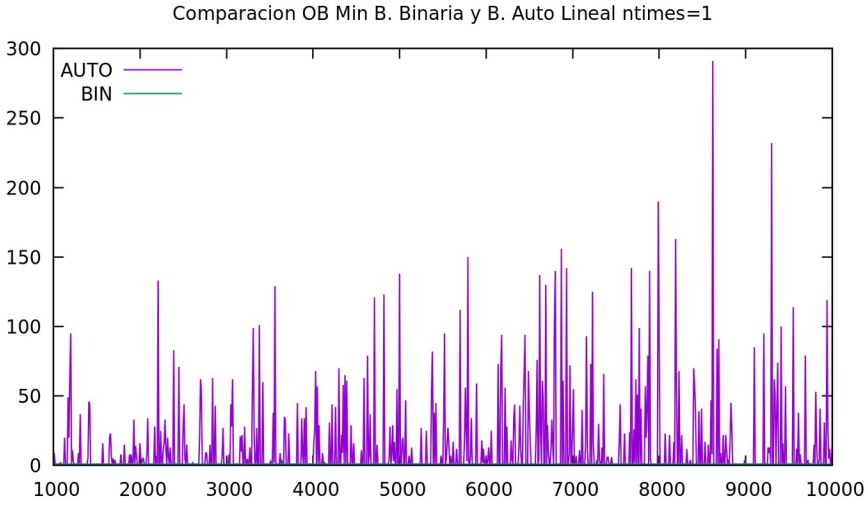


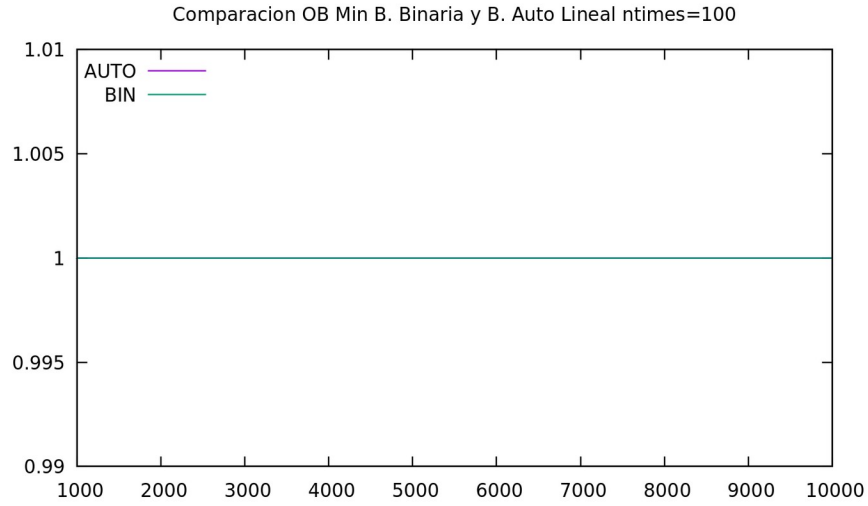


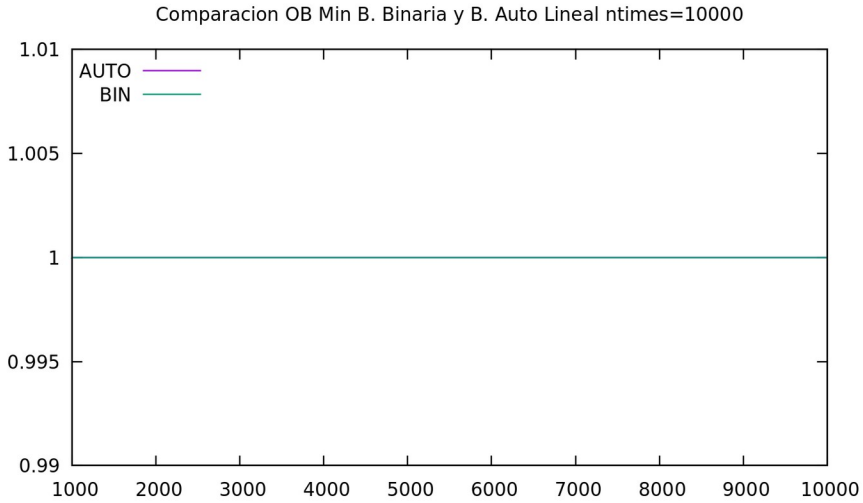


Aquí vemos que el caso peor de la búsqudea binaria sigue siendo log(n), pero en la autolineal, por mucho ntimes, sigue habiendo algún caso en el que se busca el último elemento. De ahí que salga n.

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.

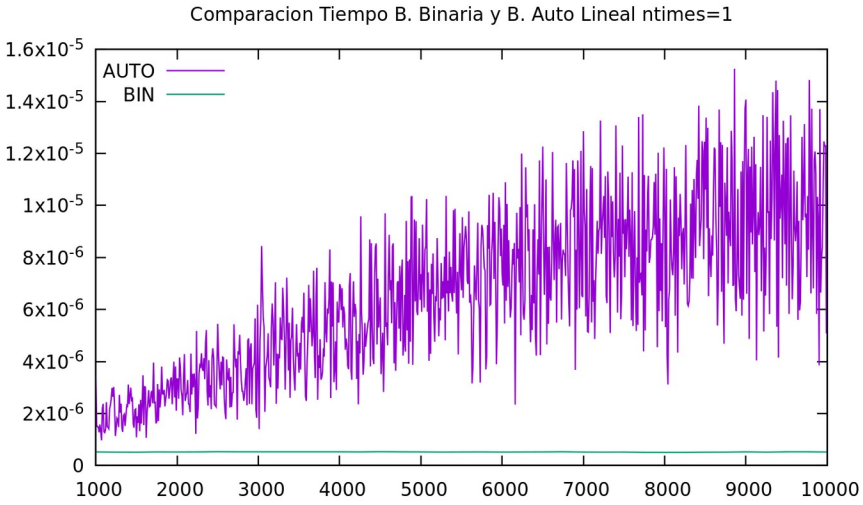


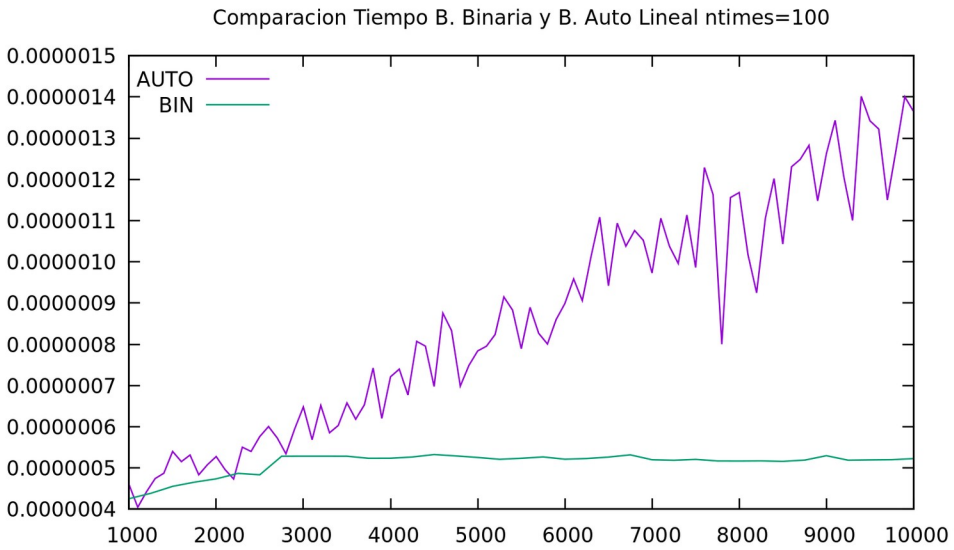


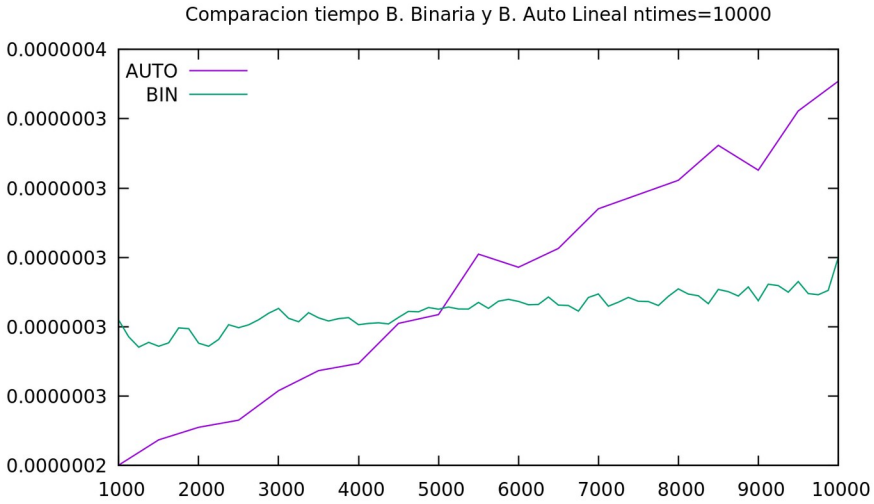


En general, el mínimo de la búsqueda binaria y lineal es de 1, salvo con alguna excepción cuando ntimes es 1. Sin embargo, cuando ntimes es muy grande, será mucho más común este 1 en auto_lineal que en bin_search.

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.







Aquí vemos las diferencias de tiempos, en general el comentario es el mismo que en las operaciones básicas medias. Sin embargo, por algún motivo aquí la búsqueda autolineal es mejor para los primeros casos, algo que no tiene sentido y que probablemente se trate de algún error al calcular los tiempos en el propio programa.

5. Respuesta a las preguntas teóricas.

5.1 Pregunta 1

La operación básica en las tres funciones es una comparativa, viendo si la clave proporcionada es igual a distintos elementos dentro de la lista. En los tres casos esta representado dentro de un “if” de modo: if(table[i]==key) …

5.2 Pregunta 2

Bin_search

El peor caso para la búsqueda binaria es buscar un elemento para el que hayas tenido que dividir la tabla en 2 tantas veces hasta que solo quede un último elemento. Estos elementos son por ejemplo, el primer elemento, el último elemento, el elemento antes y después del elemento del medio etc. El número de veces que hay que dividir un array entre 2 hasta que solo quede un elemento es claramente log(n). Por lo tanto:

$$WSSbs=O(log(N))$$

Por su parte, el mejor caso de la búsqueda binaria es el elemento del medio, puesto que al hacer la primera comprobación con el elemento del medio para proseguir buscando en según qué sub-tabla ya se encuentra el elemento. Por lo tanto, claramente:

$$BSSbs=O(1)$$

Lin_search

El peor caso para la búsqueda lineal es el último elemento, puesto que habrá que recorrer todo el array hasta llegar al elemento. Por lo tanto, habrá que hacer una comprobación para cada elemento del array.

$$WSSls=O(N)$$

Por otro lado, el mejor caso es buscar el primer elemento del array.

$$BSSls=O(1)$$

5.3 Pregunta 3

Debido al procedimiento de reemplazo, las claves mas buscadas lentamente se van moviendo mas y mas al comienzo de la lista, facilitando su ubicación. Las claves más buscadas se juntan cerca del comienzo de la lista, mientras que aquellas menos buscadas tienden a quedarse hacia el final en cada reemplazo. Esto cabe resaltar solo ocurre porque no es uniforme la distribución; en caso contrario la variación de las posiciones sería más cercana al azar, dependiendo enteramente del orden en que se buscan los elementos.

5.4 Pregunta 4

Supongamos que tenemos un array de n elementos en el que ya se han realizado varias búsquedas y los elementos más recurrentes están al principio. Con cualquier búsqueda lineal normal los el coste sería O(n), porque así funciona la búsqueda lineal. Pero la función de “potential key generator” tiene un funcionamiento peculiar, ya que vamos a buscar con mucha más frecuencia los elementos del principio. De esta manera, como mayoritariamente vamos a buscar elementos que están al principio se puede decir que el coste de ejecución es O(1). En nuestro ejemplo, cuando lo realizamos con “n_times = 10.000” el tamaño del array es muy superior al del tamaño máximo de la clave que se busca. Así conseguimos que nos quede una gráfica con una recta con tan poca pendiente. Cuanto mayor sea n_times mayor es esta diferencia y consecuentemente

menor será la pendiente de la recta. De ahí que la recta pase de ser O(n) a ser O(1). Sin embargo, esto es bastante teórico y llevado a la práctica se vuelve algo hasta imposible.

5.5 Pregunta 5

Lin_search simplemente recorre toda la lista hasta encontrar el elemento buscado; eso conlleva a una eficiencia de O(1) en el mejor caso, pero de O(N) en el peor. Bin_search corrige esto mediante el método “divide y vencerás” que implementa mediante recursión. Utilizando el hecho de que la lista esta ordenada, va dividiendo en mitades como un árbol binario, con lo que la eficiencia en el peor caso es de solamente O(logN), ya que no tiene que recorrer mucho de la lista donde sabe que no se va a encontrar la clave buscada.

6. Conclusiones finales.

Una vez finalizada la práctica podemos determinar que los objetivos de la misma se han cumplido de manera muy satisfactoria. Se han realizado todos los objetivos y se han respondido todas las preguntas de forma coherente a los resultados obtenidos. Los resultados obtenidos tanto en “Lin_Search”, “Bin_Search” como en “Lin_auto_seach” se ajustan mucho a los teóricos, con la excepción de algún caso en Bin_Search.

Se ha continuado usando los mismos entornos de trabajo que para la práctica anterior que han seguido siendo adecuados y cómodos para cada miembro de la pareja (Linux, Ubuntu, VSC, Gnuplot, GitHub...). El reparto de la tarea ha sido equitativo y ambos miembros de la pareja han trabajado satisfactoriamente. Cada miembro de la pareja se ocupó específicamente en el trabajo de una de las funciones de ordenación pero aun así el otro miembro se interesó por el trabajo del compañero