Análisis de Algoritmos 2023/2024

Práctica 3

Iker González Sánchez, Grupo 1261

1. Introducción.

En este trabajo se presenta el desarrollo y el análisis de algoritmos de búsqueda sobre diccionarios. Se utiliza un tipo abstracto de datos definido en el fichero search.h, que implementa un diccionario mediante una tabla (ordenada o desordenada). Además se comparan diferentes métodos de búsqueda, como la búsqueda binaria, la búsqueda lineal y la búsqueda lineal autoorganizada.

2. Objetivos

2.1 Apartado 1

En este apartado se crea e implementa un tipo abstracto de datos, además de sus respectivas funciones.

2.2 Apartado 2

Este apartado se enfoca en las búsquedas sobre el diccionario implementado mediante una tabla ordenada o desordenada. Se comparan diferentes métodos de búsqueda, como la búsqueda binaria, la búsqueda lineal y la búsqueda lineal autoorganizada, que adapta la posición de las claves según su frecuencia de consulta. Se miden el tiempo y el número de operaciones básicas que requieren cada uno de estos métodos en función del tamaño del diccionario y de la distribución de las claves a buscar. Se utilizan funciones generadoras de claves con distribuciones uniforme y potencial para simular diferentes escenarios de búsqueda. Se discuten los resultados obtenidos y se contrastan con las predicciones teóricas.

3. Herramientas y metodología

El entorno de desarrollo seleccionado ha sido linux-ubuntu, haciendo uso de VSCode y el terminal.

3.1 Apartado 1

Han sido implementadas numerosas funciones, por lo que se explicará su metodología para cada una de ellas:

- init_dictionary:

Reserva memoria para la estructura del diccionario y su tabla de enteros.Inicializa las propiedades del diccionario, como el tamaño, el orden y el número de datos. Y acaba devolviendo un puntero al diccionario inicializado.

- free_dictionary:

Verifica la existencia del diccionario y su tabla antes de liberar la memoria, después, libera la memoria asignada al diccionario.

- insert_dictionary:

Verifica si el diccionario está inicializado y si hay espacio disponible. Inserta un nuevo elemento en el diccionario. Si el diccionario está ordenado, mantiene el orden durante la inserción. Por último, devuelve el número de operaciones básicas realizadas durante la inserción.

- massive_insertion_dictionary:

Verifica la existencia del diccionario y la validez de la lista de claves. Itera sobre la lista de claves e inserta cada una usando la función insert_dictionary. Devuelve el número total de operaciones básicas realizadas durante la inserción masiva.

- search_dictionary:

Verifica la existencia del diccionario y del método de búsqueda. Llama al método de búsqueda proporcionado para buscar la clave en la tabla del diccionario. Devuelve lo que devuelva este método.

- bin_search:

Implementa la búsqueda binaria recursiva en una tabla ordenada, siguiendo el pseudocódigo visto en teoría. Devuelve 1 si encuentra la clave y establece la posición o 2 + el resultado de la búsqueda si no encuentra la clave (esto es porque habría realizado 2 comparaciones de clave hasta el momento). Si no encuentra la clave devuelve 0.

- lin_search:

Implementa la búsqueda lineal en una tabla desordenada, siguiendo el pseudocódigo visto en teoría. Devuelve el número de comparaciones realizadas y establece la posición si encuentra la clave. Devuelve el número total de comparaciones si no encuentra la clave.

- lin_auto_search:

Utiliza la búsqueda lineal y ajusta la posición si encuentra la clave. Intercambia el elemento encontrado con el anterior para mantener el orden. Devuelve el número total de comparaciones.

3.2 Apartado 2

La función average_search_time calcula el tiempo promedio y el número promedio de observaciones de búsqueda en un diccionario. Inicializa un diccionario con datos ordenados o

desordenados, genera claves para búsqueda, realiza búsquedas y mide el tiempo, registra estadísticas y libera la memoria. La función generate_search_times utiliza average_search_time para generar datos de tiempo y observaciones para distintos tamaños de diccionario, guardando los resultados en un archivo. Reserva memoria, itera sobre los tamaños, calcula y almacena resultados, libera memoria y devuelve el estado de la operación.

4. Código fuente

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

4.1 Apartado 1

```c
PDICT init_dictionary (int size, char order){
  PDICT pdict;
  pdict = (PDICT)malloc(sizeof(DICT));
  if(!pdict) return NULL;

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

  return pdict;
}

void free_dictionary(PDICT pdict){
  if(pdict) {
    if(pdict->table) free(pdict->table);
    free(pdict);
  }

  return;
}

int insert_dictionary(PDICT pdict, int key){
  if(!pdict) return ERR;
  int obs=0;

  if(pdict->n_data == pdict->size) return ERR;

  pdict->table[pdict->n_data] = key;
  pdict->n_data++;

  if(pdict->n_data > 1 && pdict->order == SORTED) {
    int A = pdict->table[pdict->n_data-1];
    int j = pdict->n_data-2;
    while(j >= 0 && pdict->table[j] > A) {
      pdict->table[j+1] = pdict->table[j];
      obs++;
      j--;
    }
    if(j >= 0) obs++;
    pdict->table[j+1] = A;
  }

  return obs;
}

int massive_insertion_dictionary (PDICT pdict,int *keys, int n_keys){
  int i, obs = 0, aux;

  if(!pdict || !keys || n_keys < 1) return ERR;

  for(i = 0; i < n_keys; i++) {
    aux = insert_dictionary(pdict, keys[i]);
    if(aux == ERR)
      return ERR;
    else
      obs += aux;
  }

  return obs;
}
```

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

  return method(pdict->table, 0, pdict->n_data-1, key, ppos);
}


/* Search functions of the Dictionary ADT */
int bin_search(int *table,int F,int L,int key, int *ppos){
  int M;
  if(F > L) {
    ppos = NOT_FOUND;
    return 0;
  }
  M = (F+L)/2;
  if(table[M] == key) {
    *ppos = M;
    return 1;
  }
  else if(table[M] > key)
    return 2 + bin_search(table, F, M-1, key, ppos);
  else
    return 2 + bin_search(table, M+1, L, key, ppos);
}

int lin_search(int *table,int F,int L,int key, int *ppos){
  int i;
  for(i = F; i <= L; i++) {
    if(table[i] == key) {
      *ppos = i;
      return i-F+1;
    }
  }
  *ppos = NOT_FOUND;
  return i-F;
}

int lin_auto_search(int *table,int F,int L,int key, int *ppos){
  int obs, i;

  obs = lin_search(table, F, L, key, ppos);
  if(*ppos != NOT_FOUND && *ppos > F) {
    int aux = table[*ppos];
    table[*ppos] = table[*ppos-1];
    table[*ppos-1] = aux;
    (*ppos)--;
  }
  return obs;
}
```

4.2 Apartado 2

```c
short average_search_time(pfunc_search method, pfunc_key_generator generator, char order, int N, int n_times, PTIME_AA ptime){
  if (!method || !generator || !ptime || N < 1 || n_times < 1) return ERR;

  ptime->N = N;
  ptime->n_elems = n_times;
  ptime->time = 0;
  ptime->average_ob = 0;
  ptime->min_ob = -1;
  ptime->max_ob = 0;


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

  int *perm = generate_perm(N);
  if(!perm) return ERR;

  if(order == SORTED) quicksort(perm, 0, N-1);

  massive_insertion_dictionary(pdict, perm, N);

  int *keys = (int*)malloc(n_times*N*sizeof(int));
  if(!keys) return ERR;

  generator(keys, n_times*N, N);

  int i, ob, pos;
  for(i=0; i<n_times*N; i++){
    clock_t t = clock();
    ob = search_dictionary(pdict, keys[i], &pos, method);
    t = clock() - t;

    if(ob == ERR) return ERR;

    ptime->time += ((double)t);
    ptime->average_ob += ob;

    if(ptime->min_ob == -1 || ob < ptime->min_ob) ptime->min_ob = ob;
    if(ob > ptime->max_ob) ptime->max_ob = ob;
  }

  ptime->time /= n_times*N;
  ptime->average_ob /= n_times*N;

  free_dictionary(pdict);
  free(perm);
  free(keys);

  return OK;
}
```

```c
short generate_search_times(pfunc_search method, pfunc_key_generator generator, char order, char *file, int num_min, int num_max, int incr, int n_times){
  if(!method || !generator || !file || num_min < 1 || num_max < num_min || incr < 1 || n_times < 1) return ERR;

  int veces = (num_max - num_min)/incr + 1, i;
  short status = OK;

  PTIME_AA ptime = (PTIME_AA)malloc(veces*sizeof(TIME_AA));
  if(!ptime) return ERR;

  for(i=0; i<veces && status == OK; i++){
    status = average_search_time(method, generator, order, num_min + i*incr, n_times, &ptime[i]);
  }

  if(status == OK) status = save_time_table(file, ptime, veces);

  free(ptime);

  return status;
}
```

5. Resultados, Gráficas

5.1 Apartado 1

Hemos impreso por pantalla la permutación.

En este caso sale 5 porque al insertar la permutación se ordenan los números.

Con bin_search:

```
e462135@LAPTOP-995AQRBR:~/ANAL/P3$ make exercise1_test
Running exercise1
Pratice number 3, section 1
Done by: Iker González Sánchez
Group: 1261
0 5 9 3 7 1 6 8 2 4
Key 5 found in position 5 in 5 basic op.
```

Con lin_search:

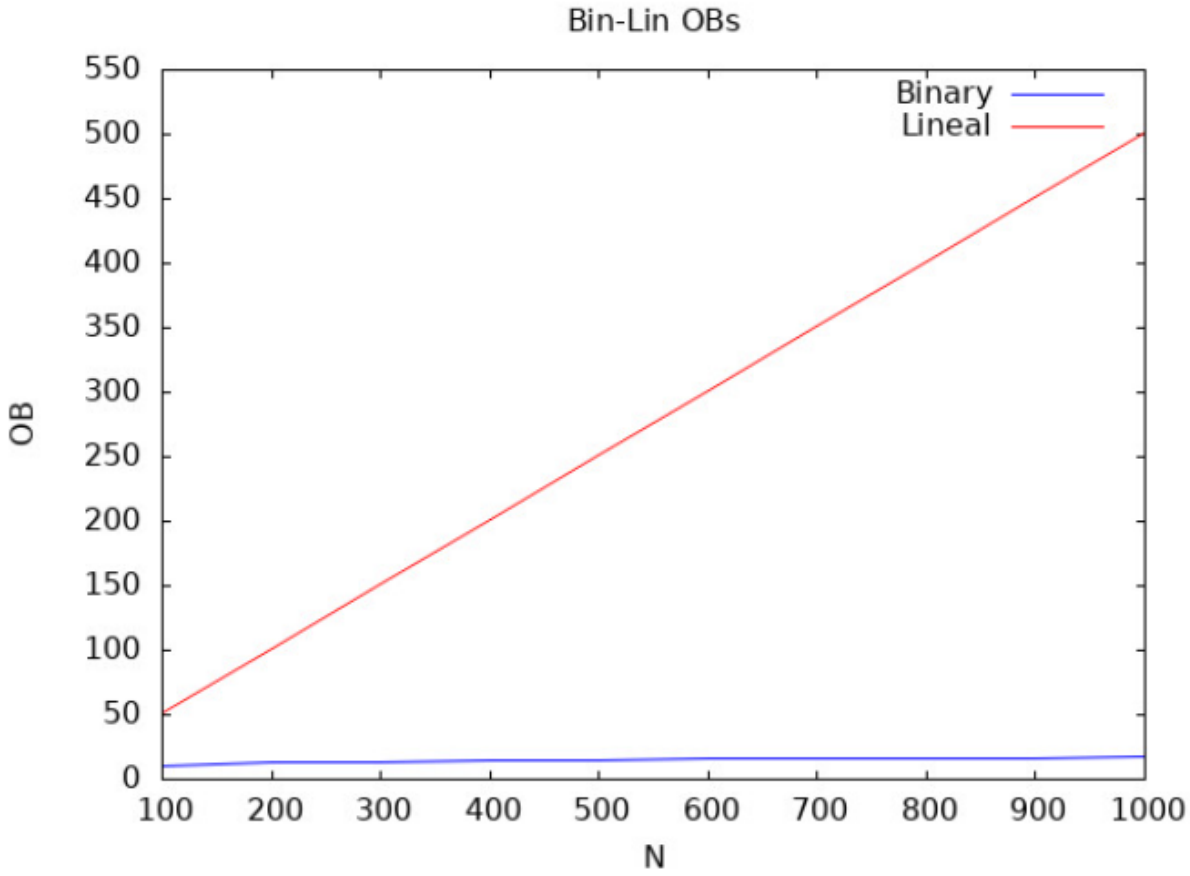
```
e462135@LAPTOP-995AQRBR:~/ANAL/P3$ make exercise1_test
Running exercise1
Pratice number 3, section 1
Done by: Iker González Sánchez
Group: 1261
9 1 3 4 8 0 2 5 6 7
Key 5 found in position 5 in 6 basic op.
```

5.2 Apartado 2

Resultados del apartado 2.

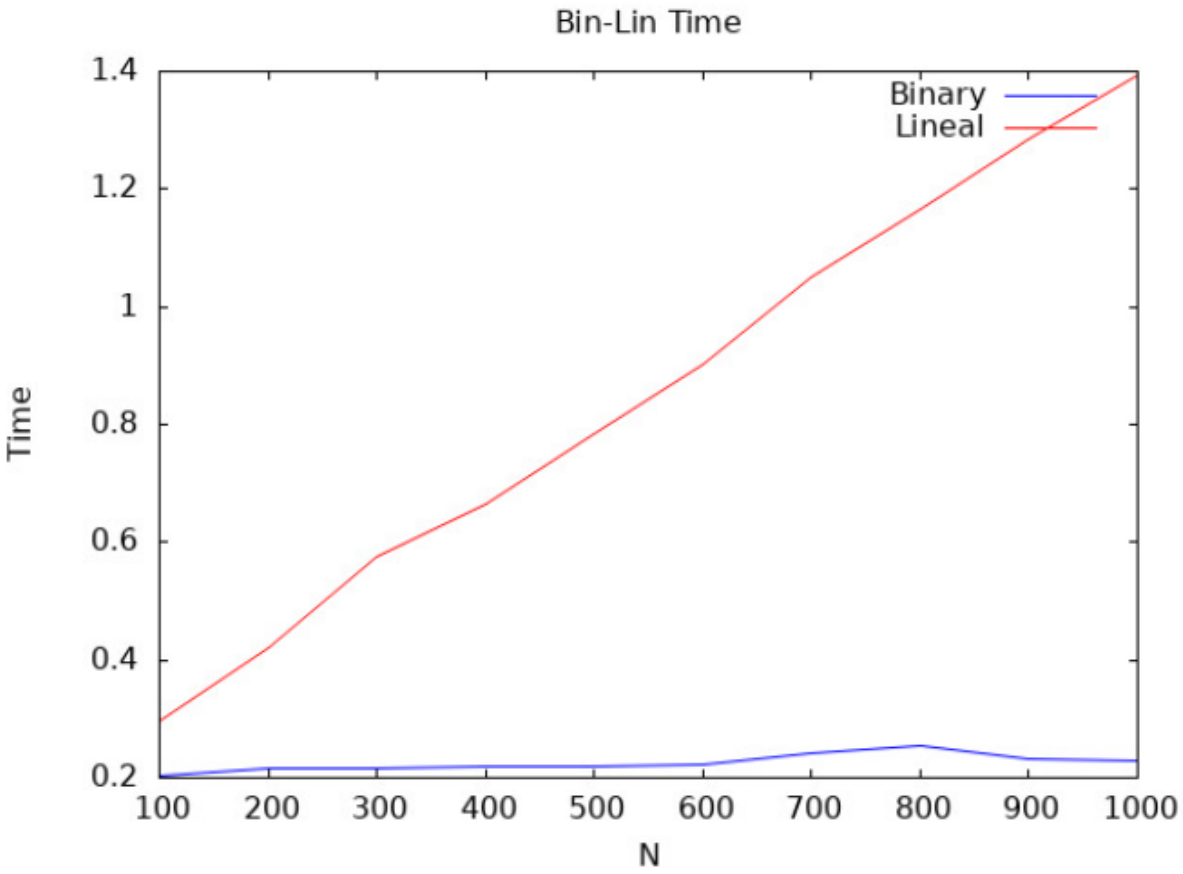
```
e462135@LAPTOP-995AQRBR:~/ANAL/P3$ make bin_exercise2_1_test
Running exercise2
Practice number 3, section 2
Done by: Iker González Sánchez
Group: 1261
Correct output
```

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



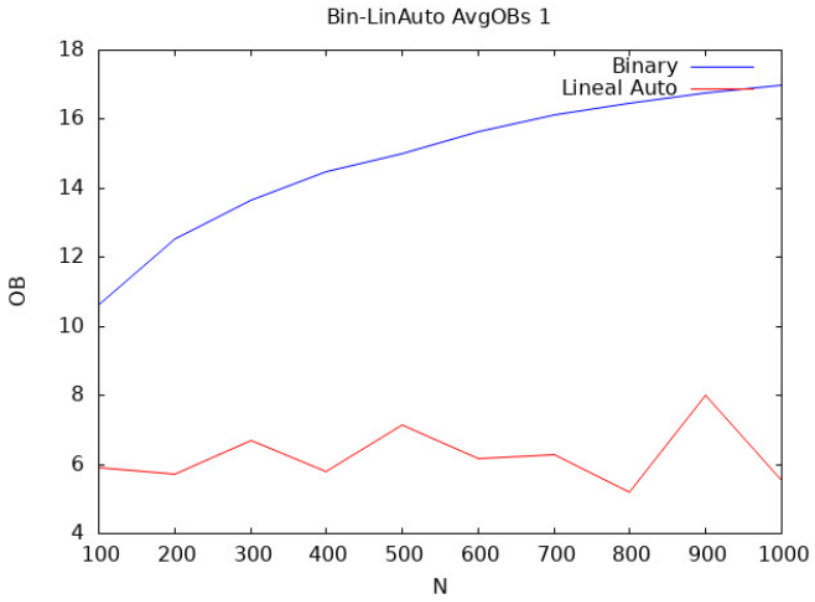
Vemos que la búsqueda binaria es logarítmica por lo que se corresponde con su coste; y la lineal también dado que es claramente lineal, es decir N.

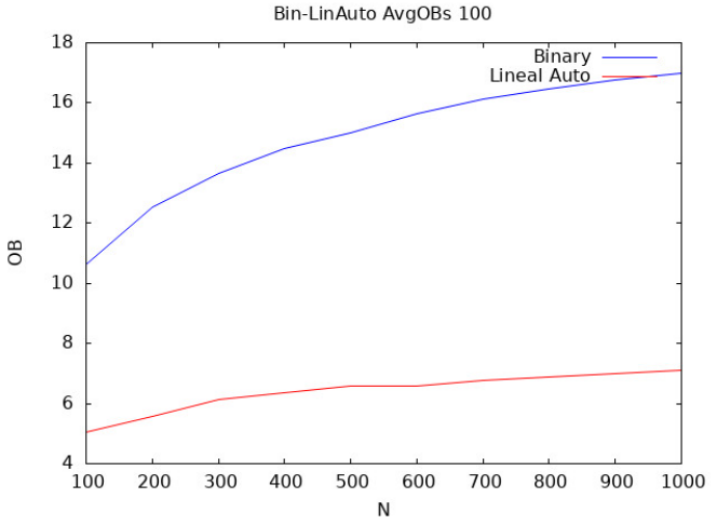
Gráfica comparando el tiempo promedio de reloj entre la búsqueda lineal y la búsqueda binaria, comentarios a la gráfica.

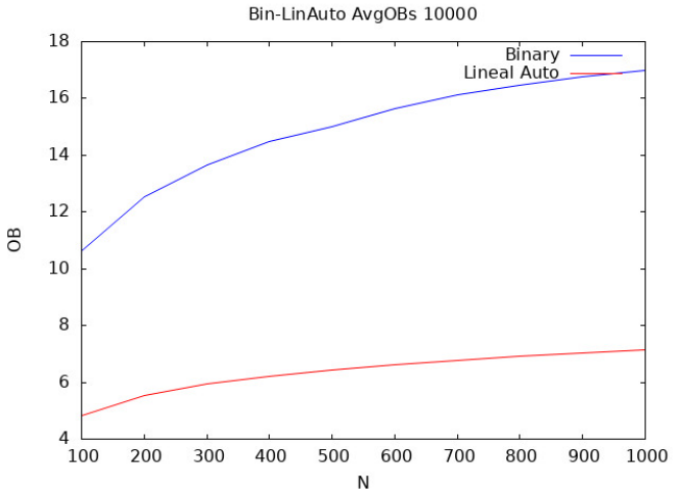


Vemos que es igual al caso anterior, está directamente relacionado con su coste teórico.

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.

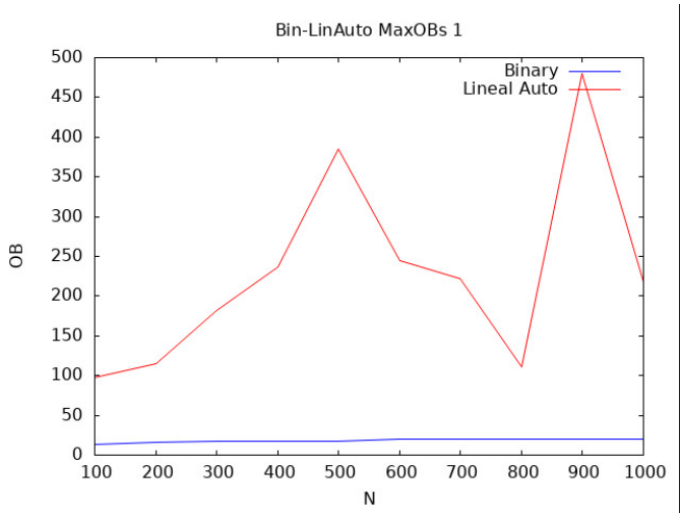

Bin-LinAuto AvgOBs 1
Binary
Lineal Auto
OB
N
18
16
14
12
10
8
6
4
100
200
300
400
500
600
700
800
900
1000

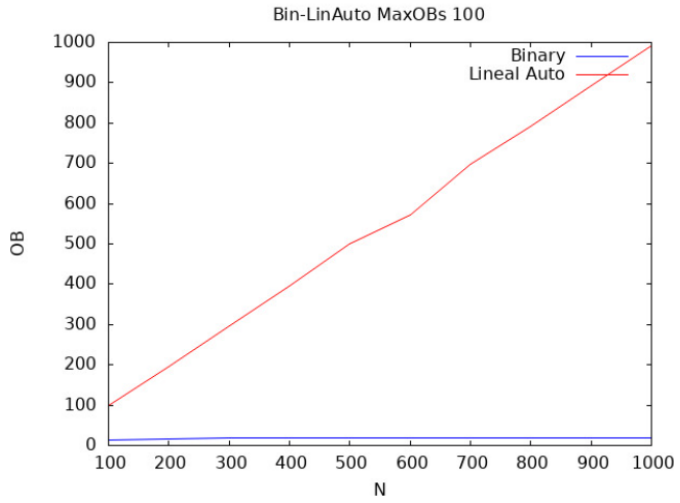

Bin-LinAuto AvgOBs 100
Binary
Lineal Auto
OB
N
18
16
14
12
10
8
6
4
100
200
300
400
500
600
700
800
900
1000

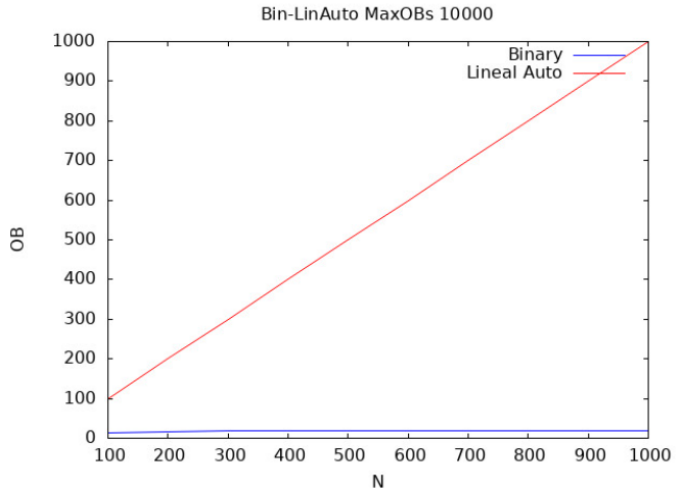

Bin-LinAuto AvgOBs 10000
Binary
Lineal Auto
OB
N
18
16
14
12
10
8
6
4
100
200
300
400
500
600
700
800
900
1000

La binaria como hemos explicado, sigue una progresión logarítmica, en cambio la autoorganizada, como va ordenando los números según sus apariciones, vemos que en la gráfica de n_times=1 hay picos, esto es debido a que haya salido algún número menos probable. Con n_times más altos vemos que es más estable aunque con menos OBs que la binaria.

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.



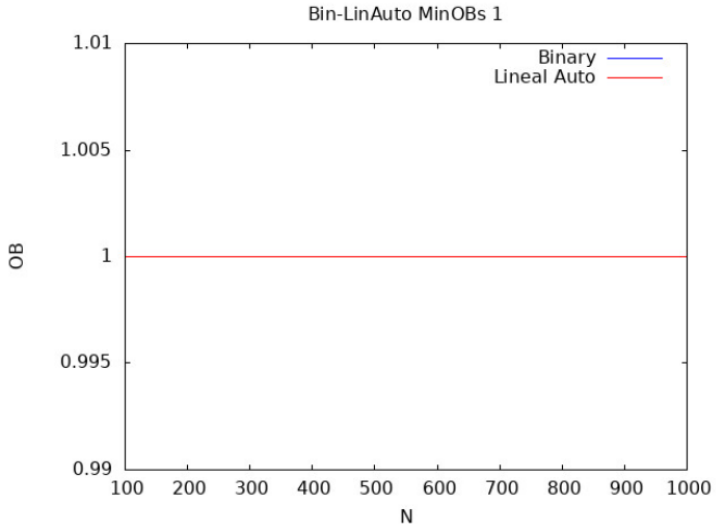


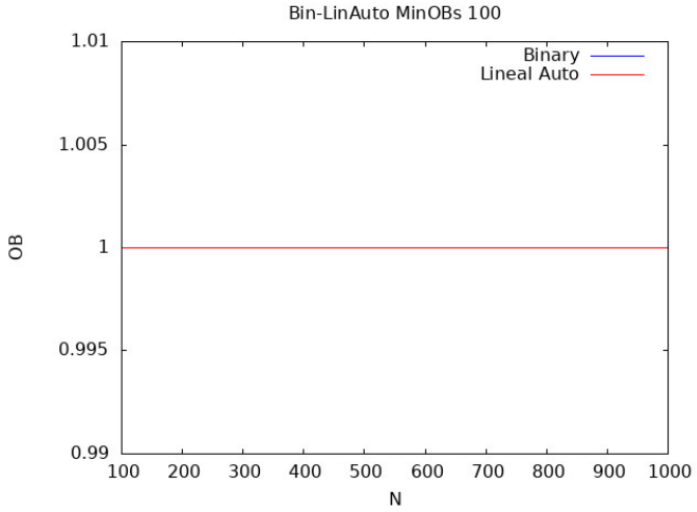


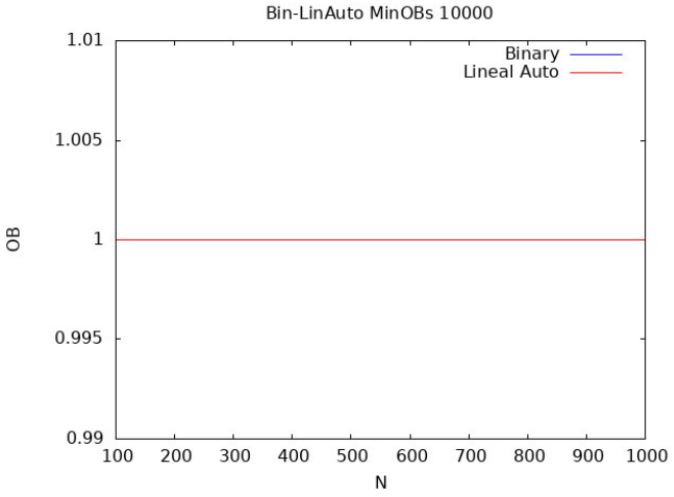
En la binaria, se obtienen valores más bajos y estables dado que su caso peor no es tan costoso debido a su método de búsqueda de dividir constantemente hace que sus OBs sean siempre similares, además sigue siendo logarítmico. En el caso de la lineal autoorganizada su caso peor tiene un crecimiento lineal, dado que siempre tiene que hacer el número máximo de comparaciones.

Para el caso de n_times=1 salen picos por lo mencionado anteriormente.

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.

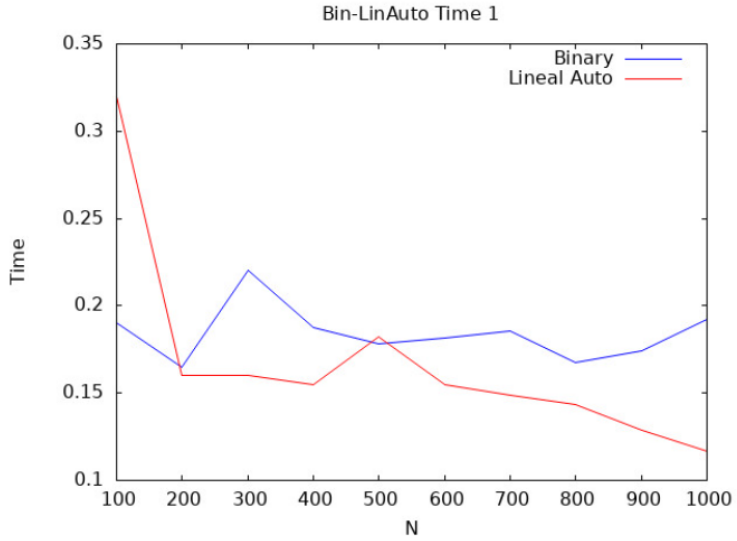


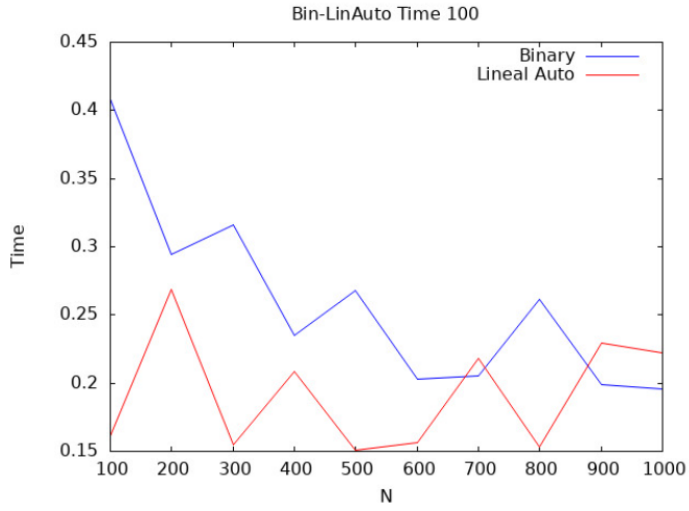


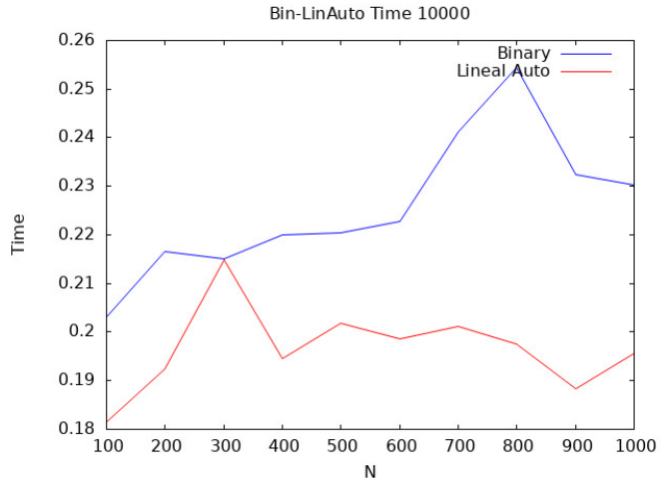


Vemos que ambas tienen siempre 1 para su caso mejor, dado que sería encontrar el número a la primera.

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.







Vemos que los tiempos son muy diferentes, esto podría ser debido a procesos del ordenador en el momento. Por lo tanto vemos que las OBs representan mejor los algoritmos.

5. Respuesta a las preguntas teóricas.

Aquí respondéis a las preguntas teóricas que se os han planteado en la práctica.

5.1 Pregunta 1

La operación básica de estos 3 algoritmos de búsqueda es la comparación de clave, como se puede ver en la implementación de nuestro código.

5.2 Pregunta 2

Para la búsqueda lineal, su mejor caso es **Θ**(1) y su caso peor **Θ**(N).

Para el caso de la búsqueda binaria, tenemos que su mejor caso es **Θ**(1) también, dado que supone encontrar el elemento buscado a la primera y su peor caso **Ω**(lg(N)) .

5.3 Pregunta 3

Las claves que más se buscan se colocan en las primeras posiciones de la tabla, ya que cada vez que buscamos una clave, esta, se mueve una posición a la izquierda.

5.4 Pregunta 4

$$A\_busq\_lin\_auto\ (N) = \sum_{i=0}^{N} Ci \cdot Pi$$

Para pocos elementos tengo una alta Pi, y un Ci € O(1)
Para los demás elementos tengo una baja Pi, y un Ci € O(n) [n/2]

Para dar un valor fijo tenemos que conocer la distribución de las Pi en nuestra función de generación de claves, y no la conocemos

Lo que sí podemos hacer, en función de esto, es aplicar dos cotas:

A_busq_lin_auto (N) = O(n/2)
A_busq_lin_auto (N) = Ω(1)

Dependiendo de la distribución de claves de búsqueda, se acercará más a su cota superior o a su cota inferior.

Si el 99% de las veces busco un 1% de los datos, estará muy cerca de coste constante.

Este caso para N=100 daría

$$A\_busq\_lin\_auto\ (N) = \sum_{i=0}^{N} Ci \cdot Pi = 0.99 \cdot 1 + 0.01 \cdot (100/2) =$$

$$= 0.99 + 0.05 = 1.04$$

5.5 Pregunta 5

Dado un conjunto ordenado S de n elementos, donde S = {a1, a2, …,an y a1<=a2<=...<=an, y dados dos índices de inicio y de fin dentro de S; para todo punto medio (inicio+fin/2) se cumple que, dada una clave a buscar que podría estar o no en S:

1) La clave buscada está en la posición medio (S[med]==k)

2) La clave buscada es menor que medio (k < S[med])

3) La clave buscada es mayor que el medio (k > S[med])

En el caso 2, la clave buscada, de existir, al ser menos que S[med],tendrá que estar entre S[inicio] yS[med-1].

En el caso 3, la clave buscada, de existir, al ser mayor que S[med],tendrá que estar entre S[med+1] yS[fin].

La búsqueda binaria acota en bucle los índices de inicio y fin como se explica en los casos 2 y 3, por lo tanto si la clave existe la acabará encontrando en algún momento en caso 1, como punto medio; o cuando se crucen los índices, que significará que no existe la clave en el conjunto ordenado.

6. Conclusiones finales.

Con lo aprendido en esta práctica hemos visto podemos decir que cada algoritmo es útil en ciertos aspectos: la binaria es la mejor para tablas ordenadas equiprobables; la lineal, en general es peor pero es más flexible al poder ser usada también con tablas desordenadas; por último la lineal autoorganizada es la mejor siempre y cuando no todos los elementos sean equiprobables.