Análisis de Algoritmos 2025/2026

Práctica 3

Rodrigo Díaz-Regañón Ureña

Daniel Martínez Fernández

Código	Gráficas	Memoria	Total

ÍNDICE

1. Introducción.

En esta práctica se pretende analizar y comparar el comportamiento de distintos algoritmos de búsqueda sobre diccionarios implementados mediante tablas (arrays). Estos algoritmos son la búsqueda lineal, la búsqueda binaria y la búsqueda lineal auto-organizada.

El trabajo consistirá en la implementación del TAD Diccionario, los algoritmos de búsqueda mencionados, y la medición experimental de su eficiencia (tiempos de ejecución y número de operaciones básicas). Finalmente, contrastaremos los resultados experimentales con los datos teóricos.

2. Objetivos.

El objetivo general de esta práctica es el estudio y análisis experimental de la eficiencia de distintos algoritmos de búsqueda sobre diccionarios implementados mediante tablas. Se busca comparar estrategias de búsqueda sobre tablas ordenadas y desordenadas (con búsqueda lineal y búsqueda binaria) frente a estrategias dinámicas adaptativas (búsqueda auto-organizada), verificando la complejidad de cada búsqueda y comprendiendo cómo la distribución de las claves influye en el rendimiento.

2.1 Apartado 1

- Implementar el TAD Diccionario, gestionando la creación, destrucción e inserción de elementos en tablas.
- Desarrollar la función insert dictionary capaz de manejar tanto tablas desordenadas (inserción al final) como tablas ordenadas (inserción manteniendo el orden).
- Implementar los algoritmos de búsqueda:
    - Búsqueda Lineal (lin_search): Para diccionarios desordenados.
    - Búsqueda Binaria (bin_search): Para diccionarios ordenados, reduciendo el espacio de búsqueda a la mitad en cada paso.
    - Búsqueda Lineal Auto-organizada (lin_auto_search): Implementando la estrategia de intercambio con el anterior para mejorar el acceso a claves frecuentes.

2.2 Apartado 2

Para la primera parte del apartado 2 se tiene como objetivo:

- Implementar las funciones de medición de tiempos average_search_time y generate_search_times para automatizar la toma de datos.
- Comparar el rendimiento (tiempo medio y número de OBs) de la búsqueda lineal sobre tablas desordenadas frente a la búsqueda binaria sobre tablas ordenadas.
- Utilizar una distribución uniforme de claves (uniform_key_generator) para validar las complejidades teóricas de los algoritmos utilizados. Para ello se utilizan tamaños de diccionario desde 1000 hasta 10000 elementos.

Para la segunda parte del apartado 2 se tiene como objetivo:

- Estudiar el rendimiento de los algoritmos bajo una distribución de claves no uniforme, donde pocas claves se buscan muy frecuentemente.
- Comparar la búsqueda binaria (en diccionario ordenado) frente a la búsqueda lineal auto-organizada (en diccionario desordenado) utilizando el generador potential_key_generator, evaluando cómo varía el rendimiento al aumentar el número de búsquedas (n_times = 1, 100, 10000), permitiendo que la lista se "auto-organice" y sitúe las claves más frecuentes al inicio.

3. Herramientas y metodología.

Para la realización de esta práctica se han usado como sistema operativo Linux (Ubuntu) y MacOS, utilizando Visual Studio Code como entorno de programación.

La compilación se realizó mediante el Makefile, utilizando el compilador gcc con las banderas -ansi -pedantic -Wall para asegurar el cumplimiento del estándar y la detección de advertencias, así como la bandera de optimización -O3 para las mediciones de tiempo.

Mediante la herramienta Valgrind se realizó un análisis completo en cada apartado para descartar cualquier tipo de problema de gestión de memoria, dado que el TAD Diccionario requiere reserva y liberación dinámica de memoria.

3.1 Apartado 1

Se comprobó el correcto funcionamiento de la implementación del TAD Diccionario y de los algoritmos de búsqueda (lin_search, bin_search, lin_auto_search) utilizando el programa exercise1.c.

Se realizaron pruebas verificando:

- La correcta creación de diccionarios ordenados (SORTED) y no ordenados (NOT_SORTED).
- El funcionamiento de la inserción masiva (massive_insertion_dictionary) y la inserción ordenada.
- Casos límite como diccionarios vacíos o claves no existentes (NOT_FOUND).
- En todos los casos se verificó con Valgrind que la función free_dictionary libera correctamente toda la memoria.

3.2 Apartado 2

En primer lugar, se utilizó el programa exercise2.c para comparar el rendimiento de la búsqueda lineal frente a la binaria bajo una distribución uniforme.

La metodología consistió en:

- Generar diccionarios de tamaños comprendidos entre 1000 y 10000 elementos.
- Utilizar el generador de claves uniformes (uniform_key_generator) buscando cada clave una sola vez (n_times = 1).
- Validar que los tiempos y el número de operaciones básicas (OBs) se ajustan a lo esperado teóricamente, O(N) para búsqueda lineal y O(log N) para búsqueda binaria.
- Se realizaron tests con casos límite (números negativos, rangos inválidos) para asegurar el correcto funcionamiento de generate_search_times.

En segundo lugar, para el análisis de la búsqueda auto-organizada, se empleó nuevamente exercise2.c pero utilizando el generador de distribución potencial (potential_key_generator) para simular un entorno de acceso sesgado (regla 80/20).

La metodología consistió en:

- Comparar la búsqueda binaria (estática) contra la búsqueda lineal auto-organizada (adaptativa).
- Variar el parámetro n_times (1, 100, 10000) para observar cómo la lista se "auto-organiza" a medida que aumentan las búsquedas, moviendo los elementos frecuentes al inicio.
- Comprobar mediante gráficas la evolución del coste medio de búsqueda a medida que el sistema se estabiliza.
- Al igual que en los apartados anteriores, se verificó la ausencia de fugas de memoria tras las ejecuciones repetitivas con Valgrind.

4. Código fuente.

4.1 Apartado 1

```
/**

* @brief Crea un diccionario vacío reservando memoria

* y controlando los errores

*

* @param size tamaño inicial del diccionario

* @param order Establece si se va a crear el diccionario ordenado o no

* @return Puntero al diccionario creado o NULL en caso de error

*/

PDICT init_dictionary(int size, char order)

{

 int i;

 PDICT dictionary = NULL;

 if ((size <= 0) || (order != SORTED && order != NOT_SORTED))

 {

   return NULL;

 }

  dictionary = malloc(sizeof(DICT));

 if (dictionary == NULL)

 {

   fprintf(stderr, "Error en la creacion del diccionario.");
```

```
  return NULL;
 }
 dictionary->n_data = 0;
 dictionary->size = size;
 dictionary->order = order;
 dictionary->table = (int *)calloc(size, sizeof(int));
 if (dictionary->table == NULL)
 {
   fprintf(stderr, "Error en la creación de la tabla del diccionario.");
   free(dictionary);
   return NULL;
 }
 for (i = 0; i < size; i++)
 {
   dictionary->table[i] = 0;
 }
 return dictionary;
}
/**
* @brief Elimina el diccionario liberando correctamente la memoria del mismo
*
* @param pdict Puntero al diccionario a eliminar
```

```
*/

void free_dictionary(PDICT pdict)

{

 if (pdict == NULL)

 {

   return;

 }

 free(pdict->table);

 free(pdict);

 return;

}

/**

* @brief Introduce el elemento clave en el diccionario

* teniendo en cuenta si este está ordenado o no

*

* @param pdict puntero del diccionario

* @param key elemento a insertar

* @return Número de operaciones básicas para

* la insercción del elemento o ERR en caso de error

*/

int insert_dictionary(PDICT pdict, int key)

{
```

```c
int aux;

int j;

int Ob = 0;



if (!pdict || pdict->size <= 0 || pdict->n_data < 0 || pdict->n_data >= pdict->size
|| !pdict->table || (pdict->order != SORTED && pdict->order != NOT_SORTED))

{

  fprintf(stderr, "Error en los parametros de entrada de insert_dictionary");

  return ERR;

}

/*guardamos al final de la tabla el elemento*/

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

if (pdict->order == SORTED)

{

  aux = pdict->table[pdict->n_data];

  j = pdict->n_data - 1;

  /* implementamos el pseudocodigo donde cada vez que se itera en el bucle se hace */

  while (j >= 0 && pdict->table[j] > aux)

  {

    pdict->table[j + 1] = pdict->table[j];

    Ob++;

    j--;

  }
```

```
   pdict->table[j + 1] = aux;

 }

 pdict->n_data++;

 return Ob;

}

/**

* @brief Realiza la inserción de varias claves

* llamando a la función insert_dictionary

*

* @param pdict Puntero al diccionario

* @param keys Tabla de elementos a insertar

* @param n_keys numero de elementos a insertar

* @return Numero de operaciones básicas realizadas

* para insertar todas las claves o ERR en caso de error

*/

int massive_insertion_dictionary(PDICT pdict, int *keys, int n_keys)

{

 int i;

 int Ob = 0;

 int aux;

 if (!pdict || pdict->size <= 0 || pdict->n_data < 0 || pdict->n_data >= pdict->size
|| !pdict->table || (pdict->order != SORTED && pdict->order != NOT_SORTED) || !keys ||
n_keys <= 0 || n_keys > (pdict->size - pdict->n_data))
```

```
 {
            fprintf(stderr,    "Error    en    los    parametros    de    entrada    de
massive_insert_dictionary\n");
   return ERR;
 }
 /* Bucle de insercion*/
 for (i = 0; i < n_keys; i++)
 {
   if ((aux = insert_dictionary(pdict, keys[i])) == ERR)
     return ERR;
   Ob += aux;
 }
 return Ob;
}


/**
* @brief Busca una una clave mediante el metodo
* dado y devolviendo su posición por un puntero
*
* @param pdict Puntero al diccionario
* @param key Clave a buscar
* @param ppos Puntero a la posición de la tabla donde se encuentra la clave
* @param method Algoritmo de búsqueda empleado
```

```
* @return Numero de OBs o ERR en caso de error

*/

int search_dictionary(PDICT pdict, int key, int *ppos, pfunc_search method)

{

 if (pdict == NULL || ppos == NULL || method == NULL)

 {

   fprintf(stderr, "Error en los parametros de entrada de search_dictionary\n");

   return ERR;

 }

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

}



/* Search functions of the Dictionary ADT */

/**

* @brief Búsqueda binaria

*

* @param table Tabla de elementos

* @param F Indice del inicio de la tabla

* @param L Indice del final de la tabla

* @param key Clave a buscar

* @param ppos Puntero a la posción donde se encuentra la tabla

* @return Número de OBs, NOT_FOUND si no se encuentra el elemento o ERR en caso de
error
```

```c
*/

int bin_search(int *table, int F, int L, int key, int *ppos)

{

 int OB = 0;

 int i;

 if (table == NULL || F > L || ppos == NULL)

 {

   fprintf(stderr, "Error en los parametros de entrada de bin_search\n");

   return ERR;

 }

 *ppos = NOT_FOUND;



 while (F <= L)

 {

   /* Calculamos el punto medio en cada iteración */

   i = (F + L) / 2;

   OB++;

   /* Comparamos */

   if (table[i] == key)

   {

     *ppos = i;

     return OB;
```

```
    }


    /* Ajustamos los límites */

    if (key < table[i])

    {

      L = i - 1;

    }

    else

    {

      F = i + 1;

    }

  }

  return NOT_FOUND;

}

/**

* @brief Búsqueda lineal

*

* @param table Tabla de elementos

* @param F Indice del inicio de la tabla

* @param L Indice del final de la tabla

* @param key Clave a buscar

* @param ppos Puntero a la posción donde se encuentra la tabla
```

```
* @return Número de OBs, NOT_FOUND si no se encuentra el elemento o ERR en caso de
error

*/

int lin_search(int *table, int F, int L, int key, int *ppos)

{

 int OB = 0, i;

 if ((table == NULL) || (F > L) || (ppos == NULL))

 {

   fprintf(stderr, "Error en los parametros de entrada de lin_search\n");

   return ERR;

 }



 *ppos = NOT_FOUND;

 i = F;

 while (i <= L)

 {

   OB++;

   if (key == table[i])

   {

     *ppos = i;

     return OB;

   }

   i++;
```

```
 }

 return NOT_FOUND;

}

/**

* @brief Búsqueda lineal autoorganizada (cuando se encuentra una clave, se intercambia con la posición

anterior, excepto si la clave encontrada ya está en la primera posición de la tabla)

*

* @param table Tabla de elementos

* @param F Indice del inicio de la tabla

* @param L Indice del final de la tabla

* @param key Clave a buscar

* @param ppos Puntero a la posción donde se encuentra la tabla

* @return Número de OBs, NOT_FOUND si no se encuentra el elemento o ERR en caso de error

*/

int lin_auto_search(int *table, int F, int L, int key, int *ppos)

{

 int i, OB = 0, aux;

 if (table == NULL || F > L || ppos == NULL)

 {

   fprintf(stderr, "Error en los parametros de entrada de lin_auto_search\n");

   return ERR;
```

```
}

 *ppos = NOT_FOUND;

  i = F;

OB++;

if (table[i] == key)

{

  *ppos = i;

  return OB;

}

i++;

while (i <= L)

{

  OB++;

  if (key == table[i])

  {

    *ppos = i;

    aux = table[i];

    table[i] = table[i - 1];

    table[i - 1] = aux;

    return OB;

  }

  i++;
```

```
 }

 return NOT_FOUND;

}
```

4.2 Apartado 2

```
/**

* @brief Subrutina para save_time_tables que guarda todos los tiempos tomados en una

* estructura TIME_AA en un archivo con un formáto en columnas

*

* @param file Nombre del archivo donde se guardaran los tiempos

* @param ptime Tabla de estrcuturas TIME_AA

* @param N Número de estructuras a escribir

* @return OK si todo va bien, ERR en caso de error

*/

short save_time_table(char *file, PTIME_AA ptime, int N)

{

   FILE *fout = NULL;

   int i;



   if (file == NULL || ptime == NULL || N <= 0)

   {

       return ERR;
```

```
    }



    if (!(fout = fopen(file, "w")))

    {

        fprintf(stderr, "Error al abrir el fichero");

        return ERR;

    }



    fprintf(fout, "%-6s %-12s %-14s %-8s %-8s\n", "N", "Time", "Average_OB", "Max_OB",
"Min_OB");



    for (i = 0; i < N; i++)

    {

        fprintf(fout, "%-6i %-12.10lf %-14.2lf %-8i %-8i\n",

                ptime[i].N,

                ptime[i].time,

                ptime[i].average_ob,

                ptime[i].max_ob,

                ptime[i].min_ob);

    }



    fclose(fout);

    return OK;
```

```
}



/**

* @brief funcion que rellena todos los campos de la estructura TIMEAA:

* - número de permutaciones

* - número de elementos por permutación

* - tiempo medio en segundos

* - número medio de OBs

* - número mínimo de OBs

* - número máximo de OBs

* @param metodo Método de ordenación

* @param generator Función que genera las claves

* @param order Si se usan las tablas ordeandas en el TAD diccionario

* @param N Tamaño del diccionario

* @param n_times Número de veces que se busca cada N clave en el diccionario

* @param ptime Puntero al diccionario

* @return OK si todo fue bien, ERR si hay algun fallo

*/

short  average_search_time(pfunc_search  metodo,  pfunc_key_generator  generator,  int
order, int N, int n_times, PTIME_AA ptime)

{

   PDICT dict;

   int *perm = NULL;
```

```
   int *keys = NULL;

   int pos;

   int n_elems;

   int i, OB_aux, min_OB = -1, max_OB = -1;

   long long sum_OB = 0;

   clock_t start, end;

   double time;



   if (!metodo || !generator || (order != SORTED && order != NOT_SORTED) || N <= 0 ||
n_times <= 0 || !ptime)

   {

         fprintf(stderr, "Los parametros introducidos en average_search_time no son
validos\n");

       return ERR;

   }



   if (!(dict = init_dictionary(N, order)))

   {

       return ERR;

   }



   if (!(perm = generate_perm(N)))

   {
```

```
        free_dictionary(dict);

        return ERR;

    }

    if (massive_insertion_dictionary(dict, perm, N) == ERR)

    {

        free_dictionary(dict);

        free(perm);

        return ERR;

    }



    n_elems = N * n_times;



    if (!(keys = (int *)calloc(n_elems, sizeof(int))))

    {

            fprintf(stderr, "Error reservando memeoria para la tabla con los N*n_times
keys\n");

        free_dictionary(dict);

        free(perm);

        return ERR;

    }



    generator(keys, n_elems, N);
```

```
start = clock();

for (i = 0; i < n_elems; i++)

{

    OB_aux = metodo(dict->table, 0, N - 1, keys[i], &pos);



    if (OB_aux == ERR || OB_aux == NOT_FOUND || dict->table[pos] != keys[i])

    {

        fprintf(stderr, "Error al realizar la busqueda");

        free_dictionary(dict);

        free(perm);

        free(keys);

        return ERR;

    }

    if (max_OB == -1)

    {

        max_OB = OB_aux;

    }

    if (min_OB == -1)

    {

        min_OB = OB_aux;

    }
```

```
    if (OB_aux > max_OB)
    {
        max_OB = OB_aux;
    }
    if (OB_aux < min_OB)
    {
        min_OB = OB_aux;
    }
    sum_OB += OB_aux;
}
end = clock();
time = (double)(end - start) / CLOCKS_PER_SEC / n_elems;

ptime->time = time;
ptime->average_ob = (double)sum_OB / (double)n_elems;
ptime->max_ob = max_OB;
ptime->min_ob = min_OB;
ptime->N = N;
ptime->n_elems = n_elems;

free_dictionary(dict);
free(perm);
```

```
   free(keys);

   return OK;

}



/**

* @brief Automatiza la toma de tiempos, llamando a generate_search_time

* con un tamaño variable que va incrementando, guarda todos los tiempos en

* dentro de un archivo

*

* @param method Método de ordenación

* @param generator Función que genera las claves

* @param order Si se usan las tablas ordeandas en el TAD diccionario

* @param file Archivo donde se guardan los tiempos

* @param num_min Tamaño mínimo por el que se empieza a tomar tiempos

* @param num_max Tamaño en el que se finaliza la toma de tiempos

* @param incr Incremento de tamaño en cada iteración

* @param n_times Número de veces que se busca cada clave en el diccionario

*/

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, n_intervals, N;
```

```
PTIME_AA ptime = NULL;



/*Comprobación de errores*/

if ((method == NULL) || (generator == NULL) || (file == NULL) ||

    (order != SORTED && order != NOT_SORTED) ||

    (num_min > num_max) || (n_times <= 0) || (incr <= 0))

{

    fprintf(stderr, "Las variables introducidas no son válidas.\n");

    return ERR;

}



/*Calcular cuántos tamaños de diccionario vamos a probar*/

n_intervals = ((num_max - num_min) / incr) + 1;

ptime = (PTIME_AA)calloc(n_intervals, sizeof(TIME_AA));

if (ptime == NULL)

{

    fprintf(stderr, "Error al reservar memoria  .\n");

    return ERR;

}



for (i = 0; i < n_intervals; i++)

{
```

```
      N = num_min + i * incr;

         if (average_search_time(method, generator, order, N, n_times, &ptime[i]) ==
ERR)

      {

          fprintf(stderr, "Error en la funcion average_search_time\n");

          free(ptime);

          return ERR;

      }

   }

   /*Guardar los resultados en el fichero*/

   if (save_time_table(file, ptime, n_intervals) == ERR)

   {

       fprintf(stderr, "Error al guardar la tabla de tiempos.\n");

       free(ptime);

       return ERR;

   }

   free(ptime);

   return OK;

}
```

5. Resultados, Gráficas.

5.1 Apartado 1

Para todos los algoritmos de búsqueda se ha usado un size de 100 y el elemento a encontrar ha sido el 67. Los resultados obtenidos tras ejecutar los siguientes algoritmos son los siguientes:

Lin_search:

```
Running exercise1
==14712== Memcheck, a memory error detector
==14712== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et al.
==14712== Using Valgrind-3.22.0 and LibVEX; rerun with -h for copyright info
==14712== Command: ./exercise1 -size 100 -key 67
==14712==
Pratice number 3, section 1
Done by: Rodrigo Diaz-Reganon Urena y Daniel Martinez Fernandez
Group: 127
Key 67 found in position 60 in 61 basic op.
==14712==
==14712== HEAP SUMMARY:
==14712==     in use at exit: 0 bytes in 0 blocks
==14712==   total heap usage: 4 allocs, 4 frees, 1,848 bytes allocated
==14712==
==14712== All heap blocks were freed -- no leaks are possible
==14712==
==14712== For lists of detected and suppressed errors, rerun with: -s
==14712== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Bin_search:

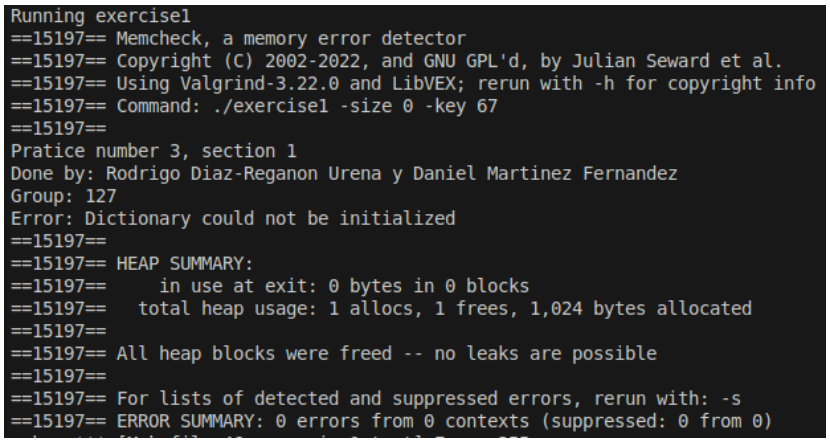
```
Running exercise1
==14860== Memcheck, a memory error detector
==14860== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et al.
==14860== Using Valgrind-3.22.0 and LibVEX; rerun with -h for copyright info
==14860== Command: ./exercise1 -size 100 -key 67
==14860==
Pratice number 3, section 1
Done by: Rodrigo Diaz-Reganon Urena y Daniel Martinez Fernandez
Group: 127
Key 67 found in position 66 in 7 basic op.
==14860==
==14860== HEAP SUMMARY:
==14860==     in use at exit: 0 bytes in 0 blocks
==14860==   total heap usage: 4 allocs, 4 frees, 1,848 bytes allocated
==14860==
==14860== All heap blocks were freed -- no leaks are possible
==14860==
==14860== For lists of detected and suppressed errors, rerun with: -s
==14860== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Lin_auto_search:

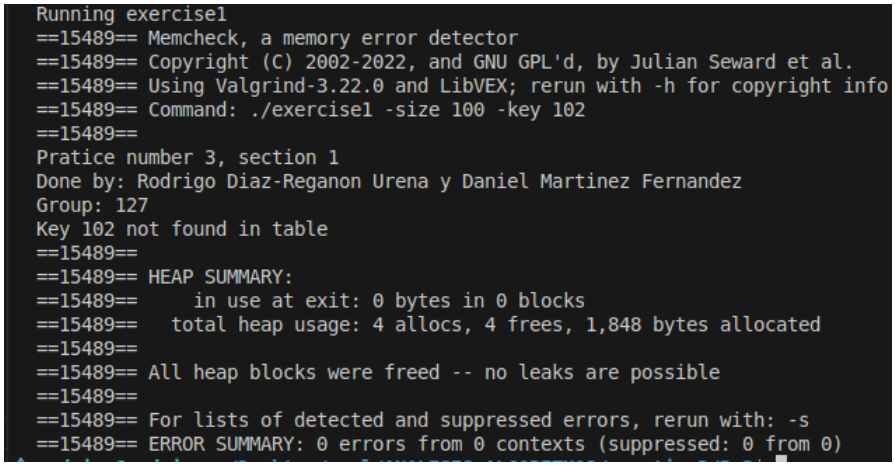
```
Running exercise1
==15004== Memcheck, a memory error detector
==15004== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et al.
==15004== Using Valgrind-3.22.0 and LibVEX; rerun with -h for copyright info
==15004== Command: ./exercise1 -size 100 -key 67
==15004==
Pratice number 3, section 1
Done by: Rodrigo Diaz-Reganon Urena y Daniel Martinez Fernandez
Group: 127
Key 67 found in position 82 in 83 basic op.
==15004==
==15004== HEAP SUMMARY:
==15004==     in use at exit: 0 bytes in 0 blocks
==15004==   total heap usage: 4 allocs, 4 frees, 1,848 bytes allocated
==15004==
==15004== All heap blocks were freed -- no leaks are possible
==15004==
==15004== For lists of detected and suppressed errors, rerun with: -s
==15004== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

También se han probado casos límite:

Diccionario vacío:



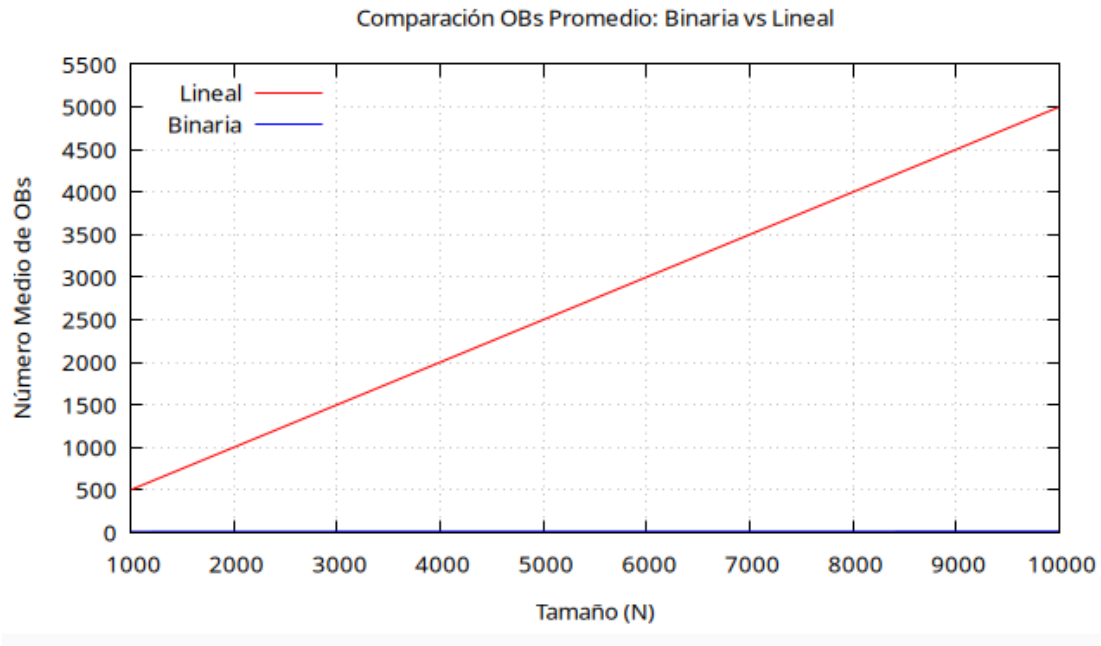
Clave inexistente:



5.2 Apartado 2

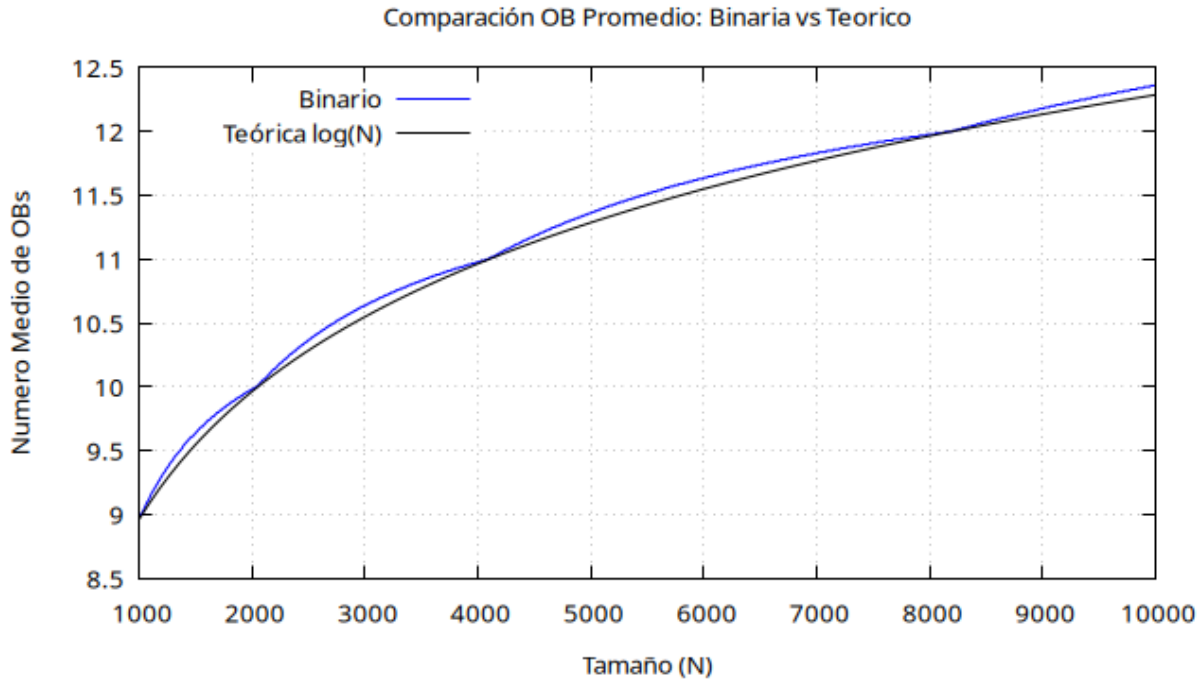
Para comprobar los resultados del apartado 2 se van a crear las siguiente gráficas:

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



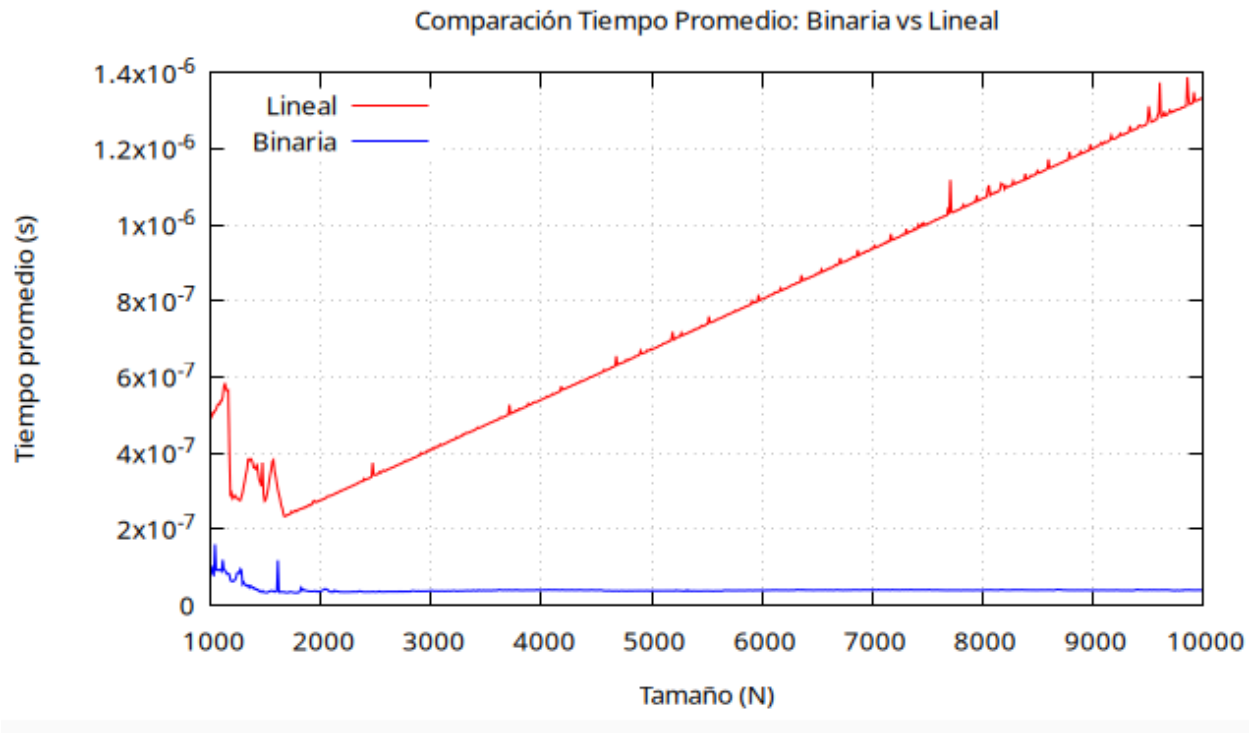
En la primera gráfica se puede observar la enorme diferencia de eficiencia entre ambos algoritmos. La línea roja, correspondiente a la Búsqueda Lineal, exhibe un crecimiento estrictamente lineal con una pendiente pronunciada, alcanzando un promedio de 5.000 operaciones básicas para un tamaño de N=10000.. Esto confirma la complejidad teórica de O(N), donde el promedio de búsquedas fallidas o exitosas en una distribución uniforme tiende a N/2.

Por el contrario, la línea azul de la Búsqueda Binaria aparece prácticamente plana y pegada al eje horizontal. Esto no indica que el coste sea cero, sino que la escala necesaria para representar el crecimiento lineal hace que el crecimiento logarítmico sea visualmente despreciable. Mientras que el algoritmo lineal requiere miles de comparaciones, el binario apenas necesita una decena, demostrando su superioridad absoluta para grandes volúmenes de datos.



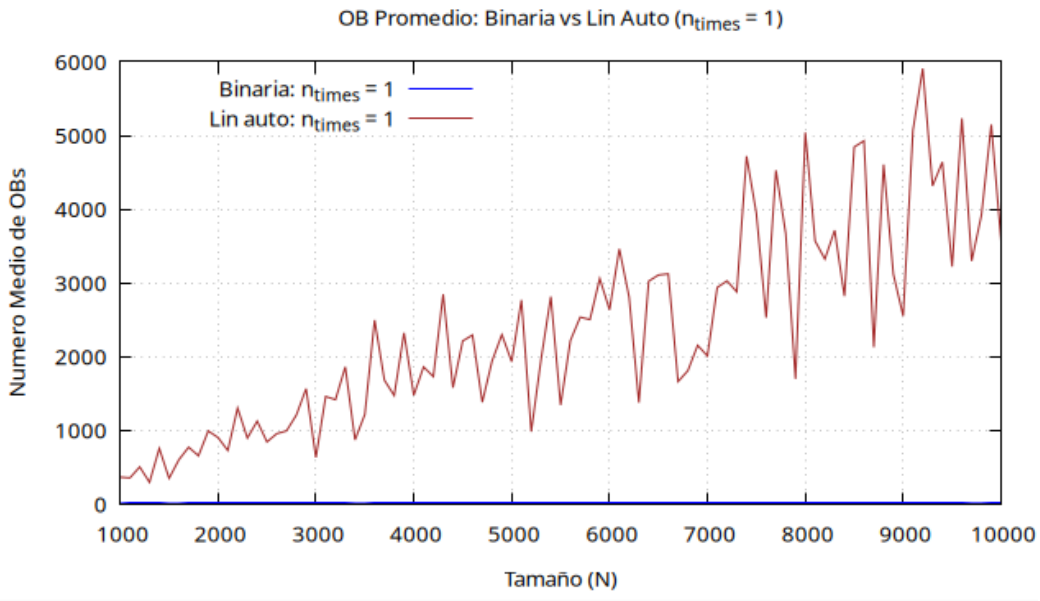
Para analizar correctamente el comportamiento de la Búsqueda Binaria, esta segunda gráfica realiza un cambio de escala. Aquí se compara la curva experimental (línea azul) con la función teórica f(x) = log2(x) - 1 (línea negra). Se observa un ajuste casi perfecto entre los datos obtenidos y el modelo teórico. Este ajuste confirma que la implementación del algoritmo es correcta y cumple con la complejidad de O(log N).

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

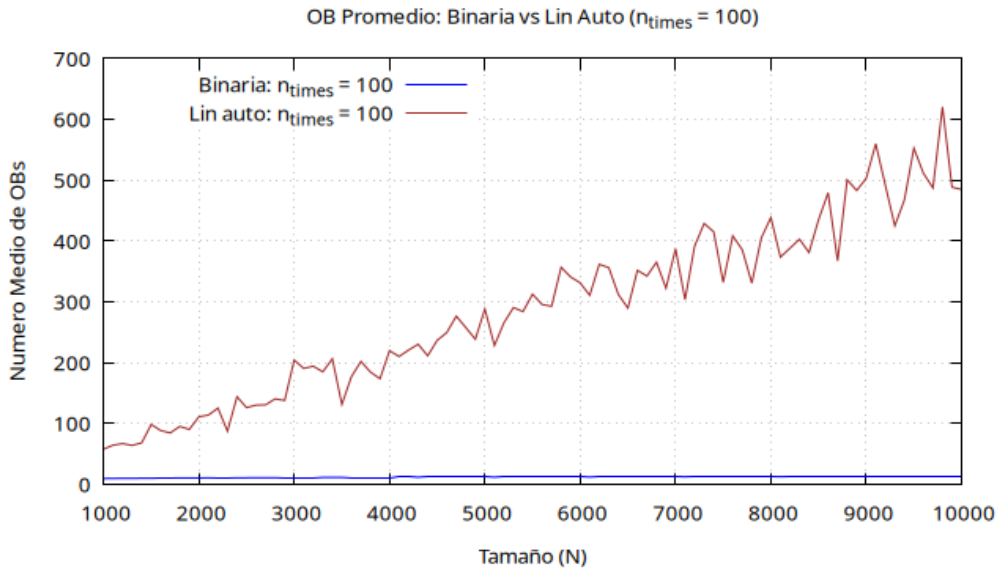


En la gráfica se observa el comportamiento temporal de ambos algoritmos medido en segundos. La Búsqueda Lineal (línea roja) presenta un crecimiento constante y proporcional al tamaño de la entrada (N), alcanzando aproximadamente $1.3 \cdot 10^{-6}$ segundos para N=10.000. Se aprecian ciertas fluctuaciones en los tamaños pequeños (N < 2000), atribuibles a la resolución del reloj del sistema y a la sobrecarga inicial del proceso, pero la tendencia general se estabiliza rápidamente en una recta. Por otro lado, la Búsqueda Binaria (línea azul) se mantiene plana en la parte inferior de la gráfica, con tiempos prácticamente despreciables en comparación con la lineal, lo que visualmente confirma su eficiencia superior sin verse afectada significativamente por el aumento de N en esta escala.

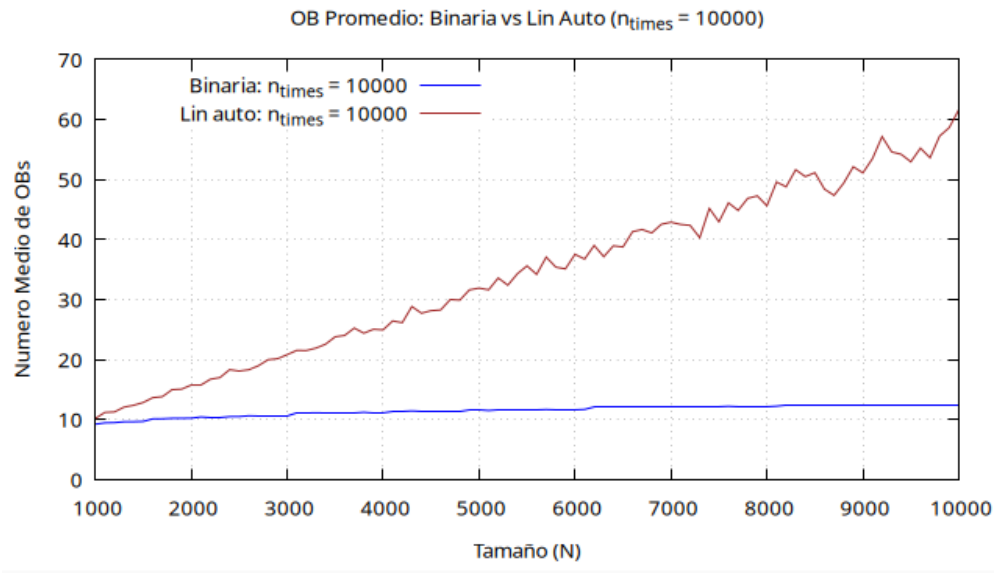
3. **Gráfica comparando el número promedio de OBs entre la búsqueda binaria y la búsqueda lineal auto organizada (para los valores de n_times=1, 100 y 10000), comentarios a la gráfica.**



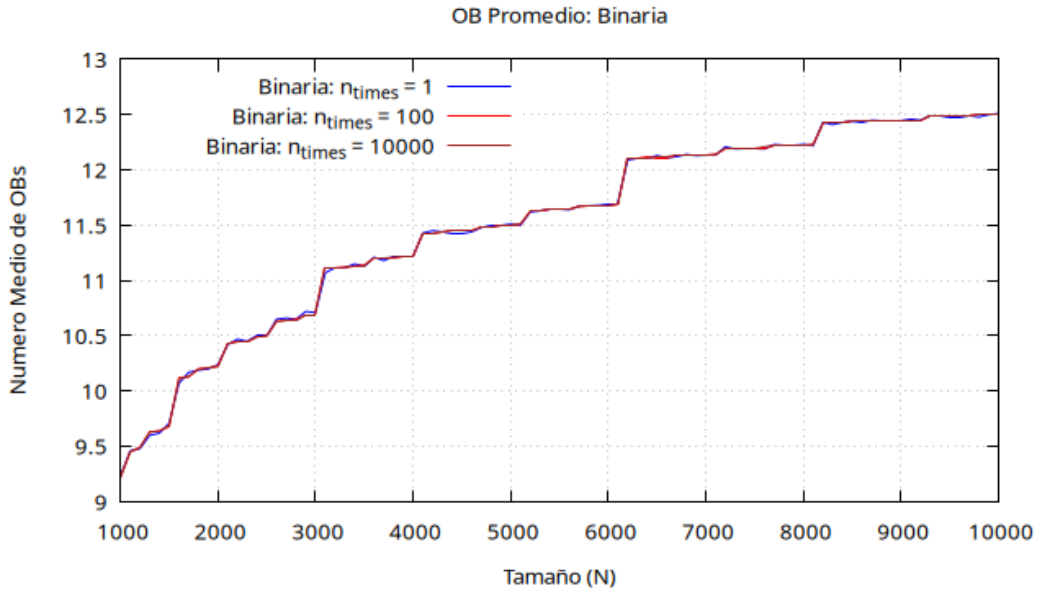
Esta gráfica compara la búsqueda binaria con la búsqueda lineal auto organizada para un valor de n_times=1. Se puede observar como la escala entre los dos algoritmos es completamente distinta. La búsqueda binaria ronda entre 9 y 13 operaciones básicas sin importar el aumento del tamaño, mientras que la búsqueda lineal auto organizada ronda entre 300 y 5000 operaciones básicas, siendo más eficiente la búsqueda binaria.



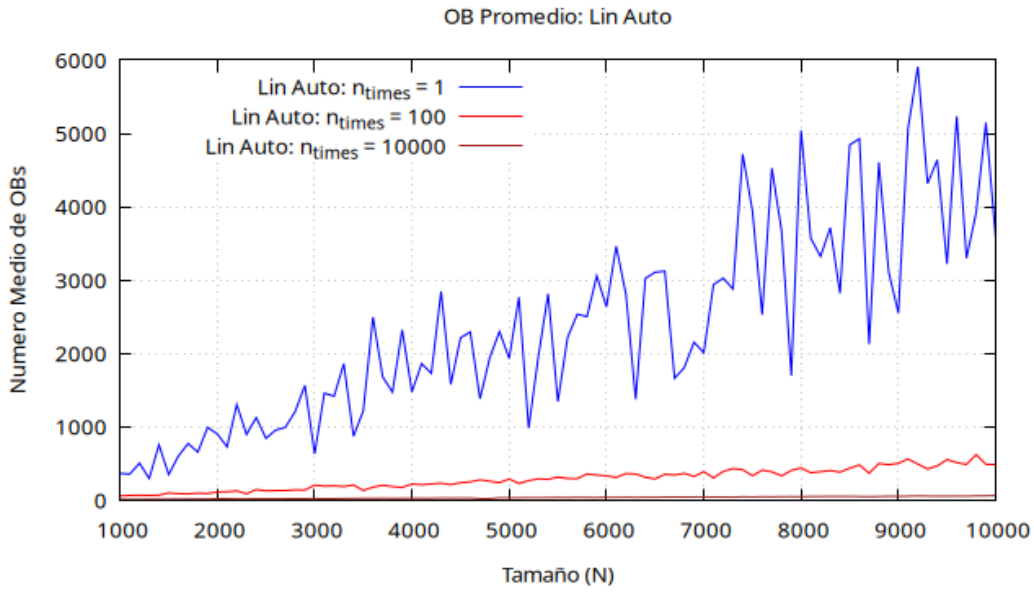
Esta gráfica compara la búsqueda binaria con la búsqueda lineal auto organizada para un valor de n_times=100. Se puede observar como la diferencia de escala entre los dos algoritmos se va acortando gracias al intercambio de elementos de la búsqueda auto organizada. La búsqueda binaria sigue rondando entre 9 y 13 operaciones básicas sin importar el aumento del tamaño, mientras que la búsqueda lineal auto organizada ahora se encuentra entre 50 y 600 operaciones básicas.



Esta gráfica compara la búsqueda binaria con la búsqueda lineal auto organizada para un valor de n_times=10000. Se puede observar como gracias al aumento del n_times y el intercambio de elementos de la búsqueda auto organizada, ambos algoritmos tienen una escala similar. La búsqueda binaria sigue rondando entre 9 y 13 operaciones básicas mientras que la búsqueda lineal auto organizada ahora se encuentra entre 10 y 60 operaciones básicas.



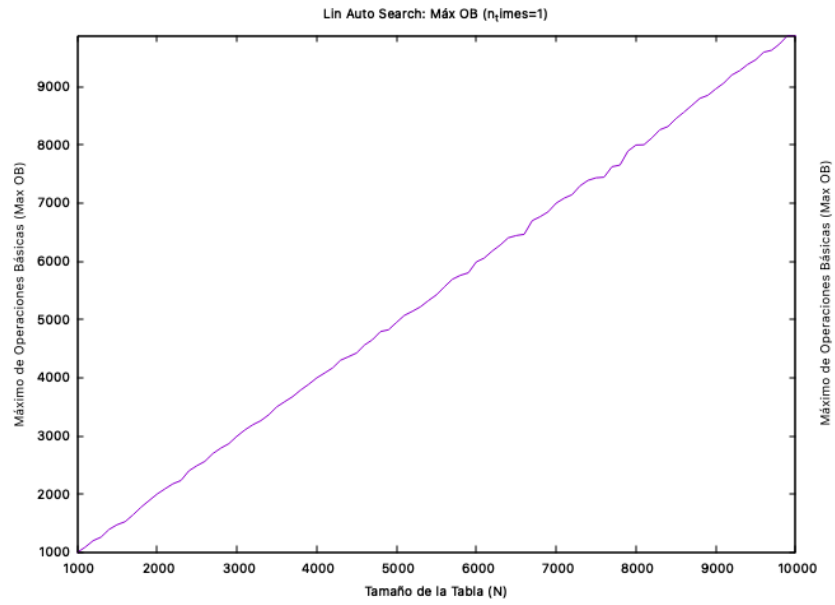
Esta gráfica muestra como el número de operaciones básicas va disminuyendo en la búsqueda lineal auto organizada a medida que aumenta el n_times, ya que los elementos más repetidos aparecen antes, de modo que se necesiten cada vez menos OBs.

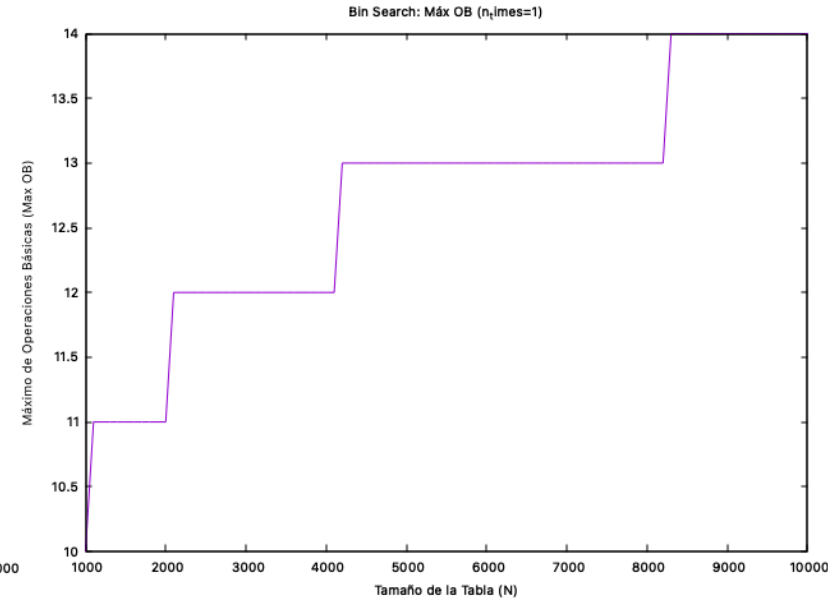


Está gráfica muestra que el aumento del n_times no afecta en la reducción de OBs en búsqueda binaria.

4. **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.**

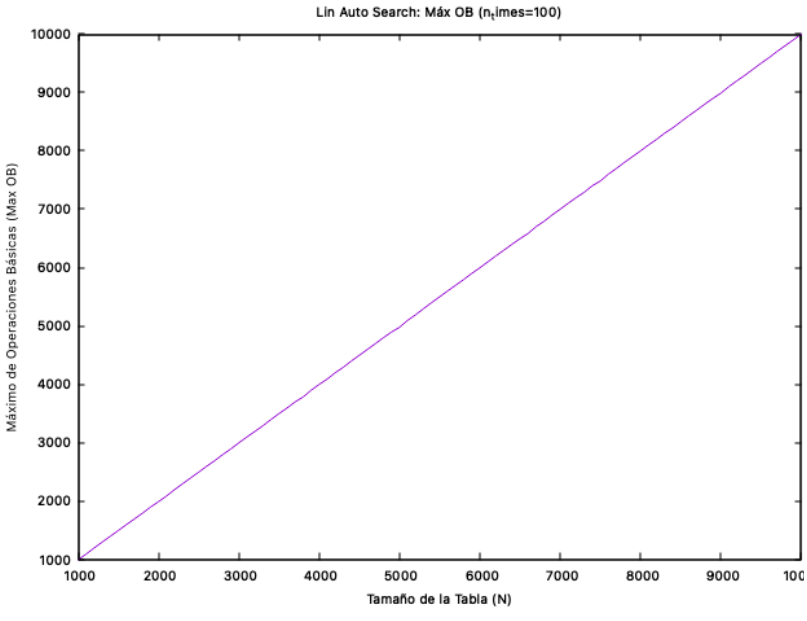
N_times = 1

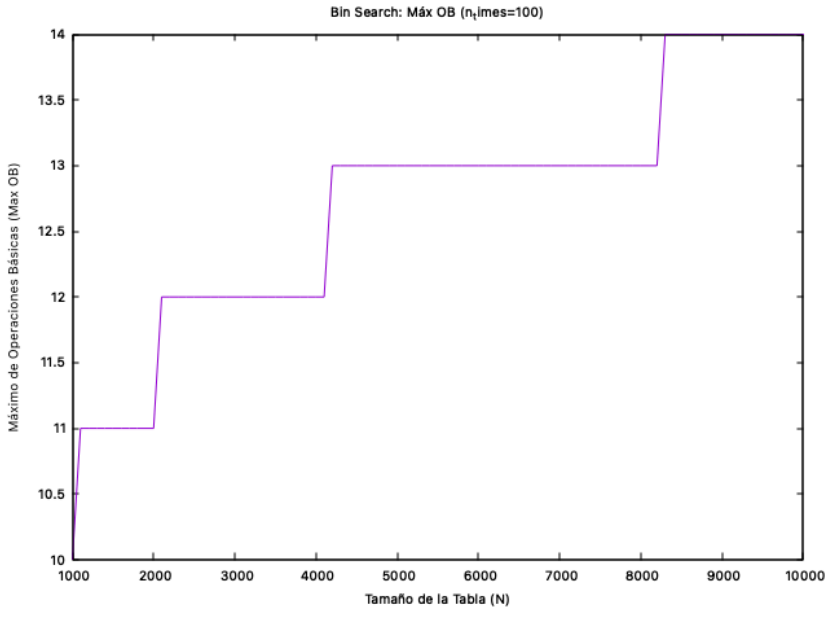




Se puede observar como la lineal-autoorganizada sigue un crecimiento lineal, ya que es muy probable que algunos de los datos con mayor probabilidad se encuentren al final de la tabla, como sigue siendo una búsqueda lineal siempre va a ser peor que la búsqueda binaria a lo que MaxObs se refiere.

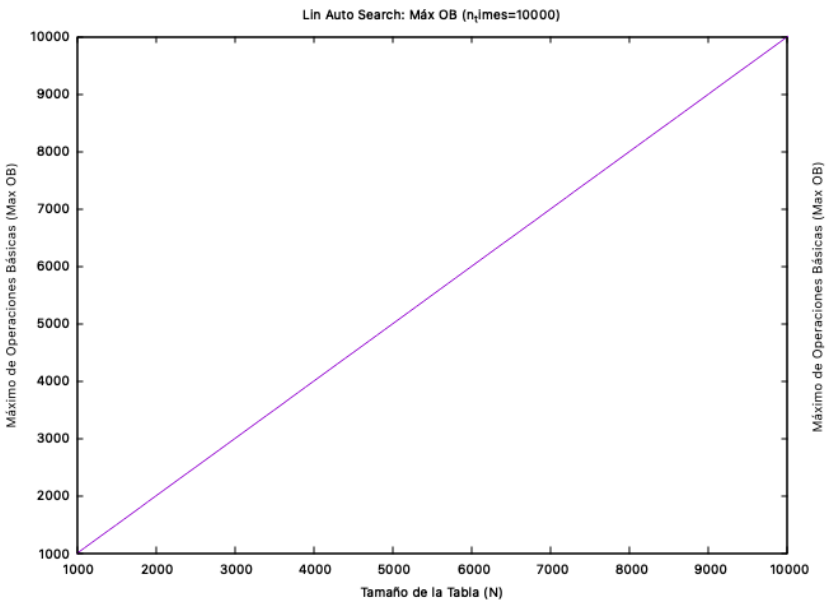
N_times = 100

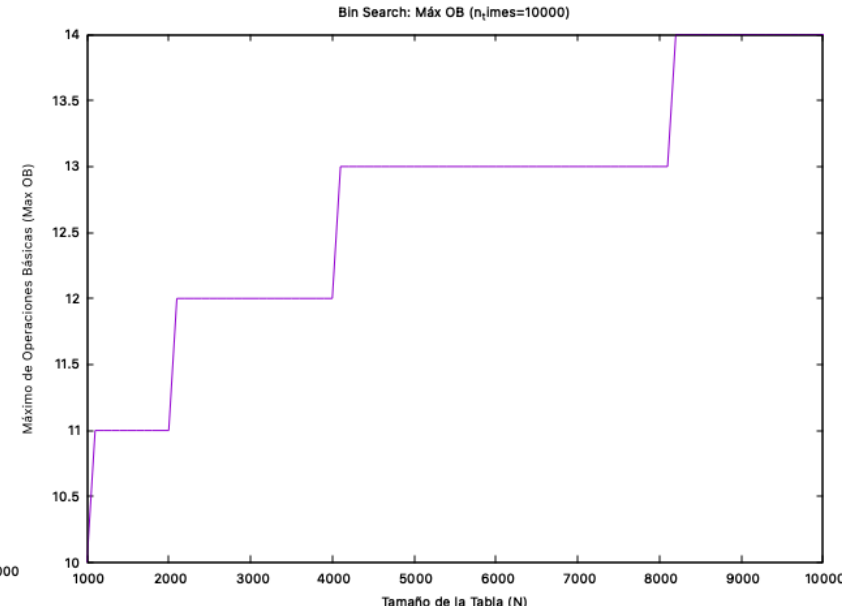




Cuando se aumenta el número de veces que se buscan elementos, se observa como la búsqueda binaria sigue quedando igual, ya que el 1 que es el elemento que más probabilidad tiene de salir, es el peor caso de la búsqueda lineal ($\lceil Log_2(N) \rceil$) Y este va a salir con un 99.999999% de seguridad. Por otro lado, la gráfica LinAuto se va a ir pareciendo cada vez más a la 100% lineal, ya que al aumentar n_times hay más probabilidad de que salga algún elemento de los finales.

N_times = 10000

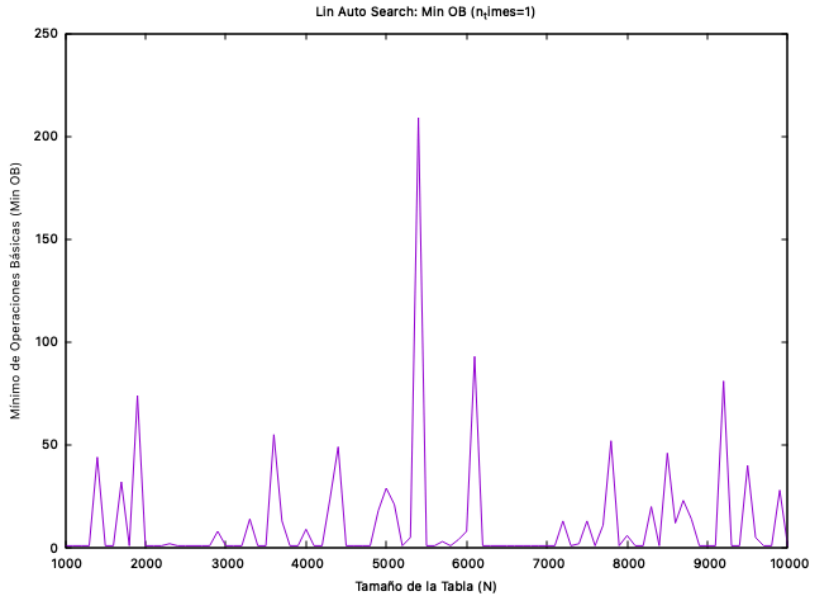


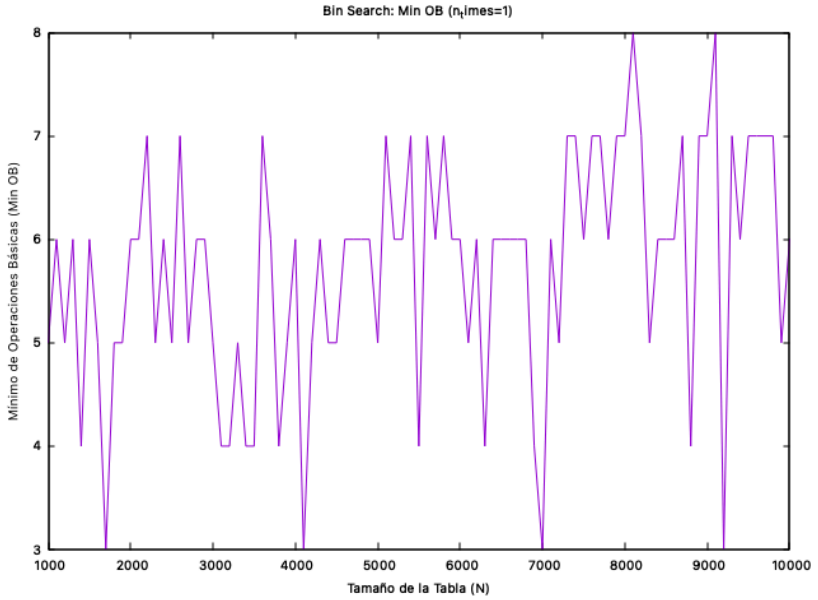


Vemos como estas gráficas siguen las mismas observaciones que las anteriores (la lineal-autoorganizada es como la lineal prácticamente y la binaria sigue igual) por lo que podemos concluir con que la búsqueda binaria es sólidamente mejor en MaxObs que la Lineal Autoorganizada.

5. **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.**

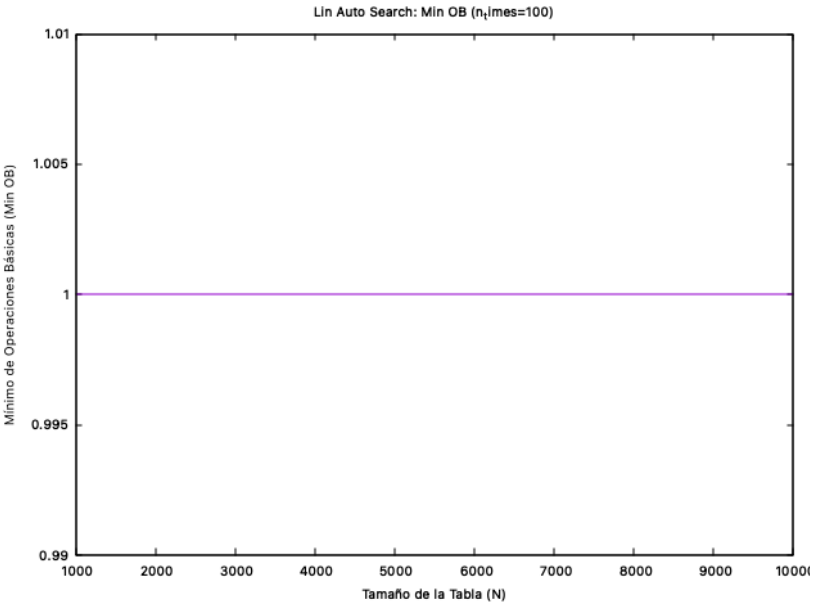
N_times = 1

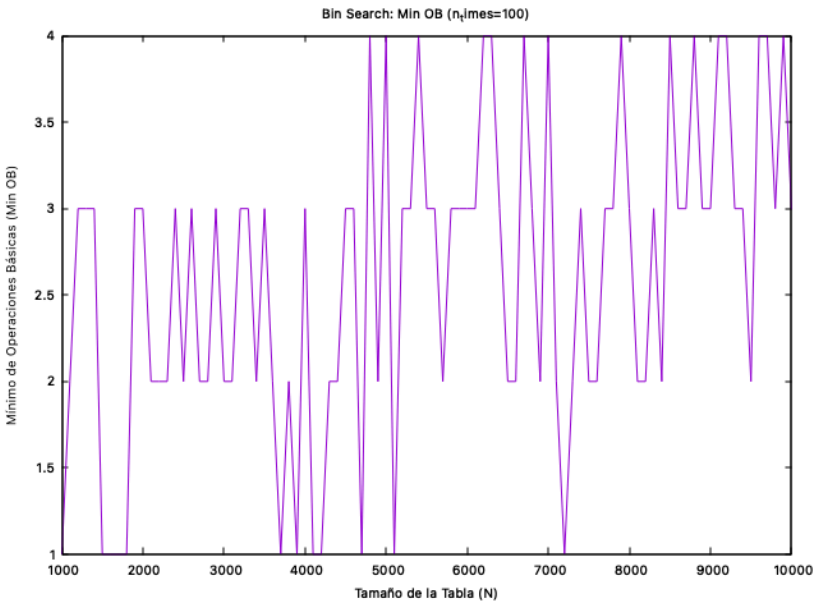




La Búsqueda Lineal Auto-organizada muestra un comportamiento muy ruidoso con picos altos (hasta aprox 220 OBs), indicando que con solo una repetición, el Mínimo de OBs encontrado puede ser alto. En contraste, la Búsqueda Binaria tiene un Mín OB estable y bajo (entre 1 y 8 OBs) con ruido, mostrando que su estructura es inherentemente más eficiente en el mejor caso, esto se debe a que para llegar al mejor caso se necesita buscar un elemento del medio que no es muy probable

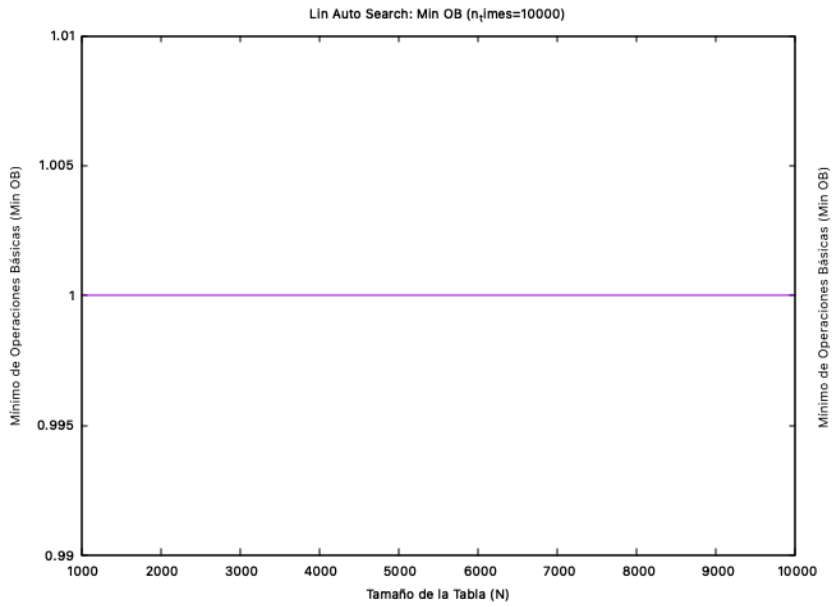
N_times = 100

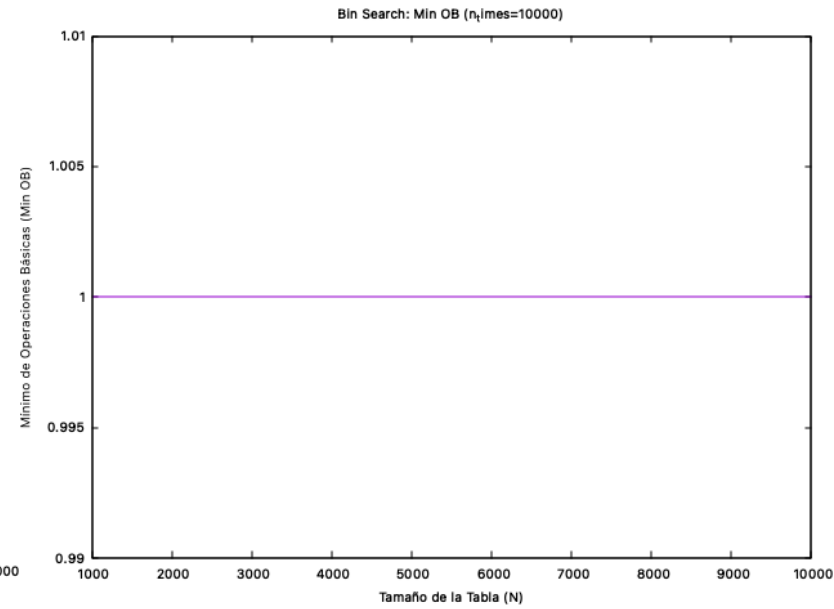




Ambas curvas se estabilizan drásticamente. La Lineal Auto-organizada converge a 1 OB, lo que demuestra que con 100 búsquedas, la clave más probable ha sido movida a la primera posición, logrando el mejor caso O(1). La Binaria también se mantiene constante y baja (entre 1 y 4 OBs), ya que a mayor n_times más probabilidad hay de que salga un elemento del medio.

N_times = 10000

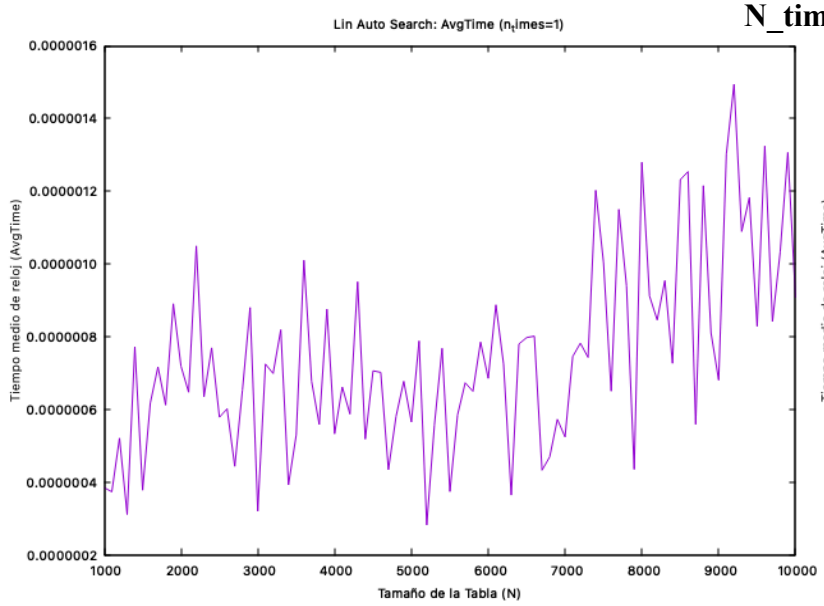


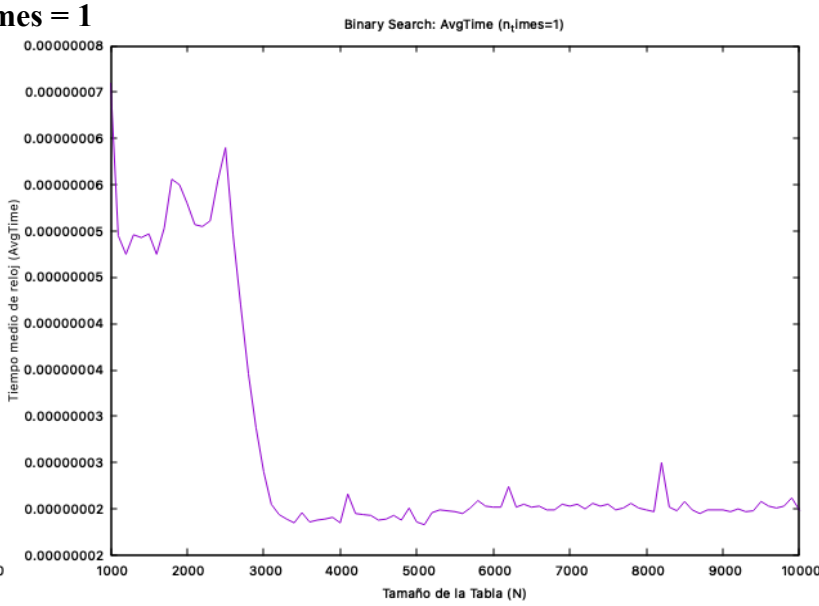


Ambas curvas alcanzan la estabilidad máxima. La Lineal Auto-organizada se fija en 1 OB, confirmando el éxito de la heurística de auto-organización bajo una distribución de claves sesgada. La Búsqueda Binaria también se mantiene constante y mínima, demostrando que N_times = 10000 es suficiente para que se haga al menos una búsqueda del elemento del medio.

6. **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.**

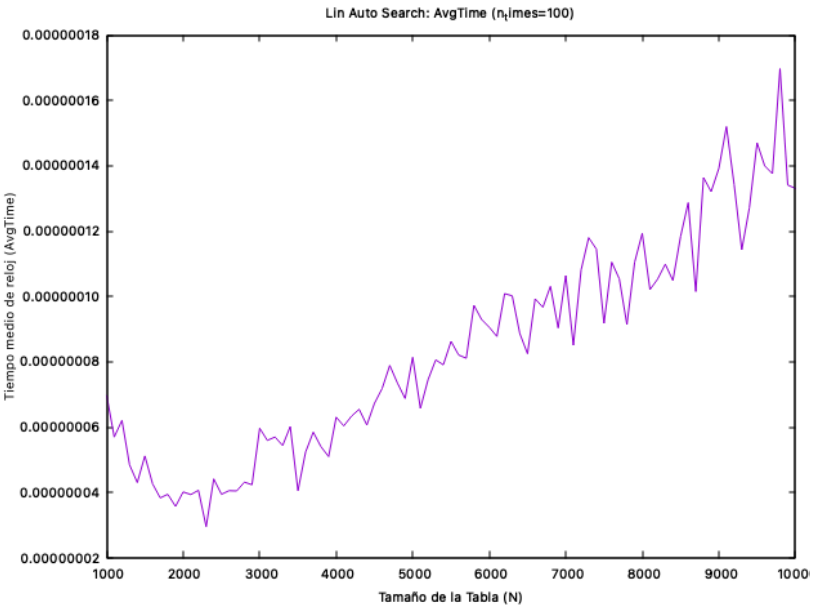
N_times = 1

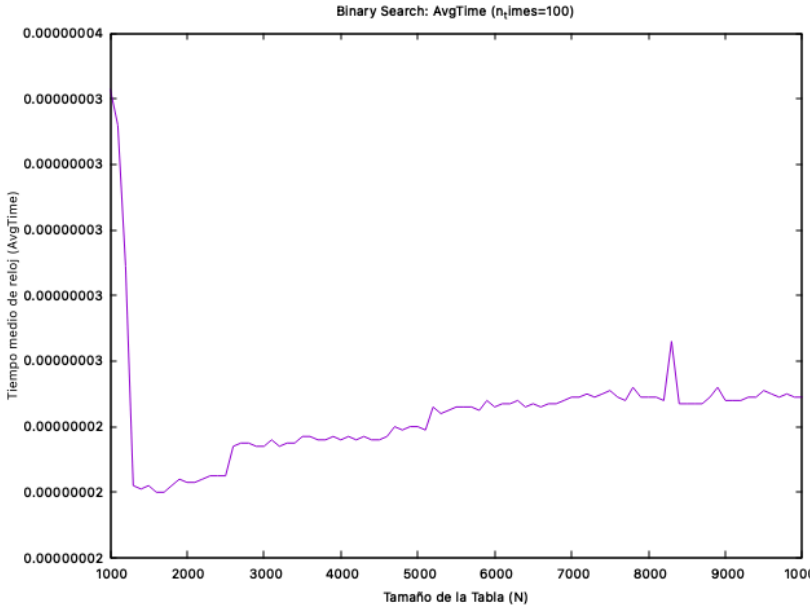




Ambas gráficas muestran un tiempo de ejecución muy ruidoso y errático, lo cual es típico al medir el tiempo de reloj con muy pocas repeticiones. Los valores de la Búsqueda Binaria tienden a ser ligeramente más bajos que los de la Lineal Auto-organizada. No obstante, se observa como el tiempo de la binaria cae drásticamente que aunque se desconoce al 100% a que se puede deber, se intuye que sea algo del ordenador. Específicamente, estas caídas bruscas pueden ocurrir cuando el tamaño de la tabla (N) cruza ciertos umbrales que afectan la eficiencia con la que la memoria se accede.

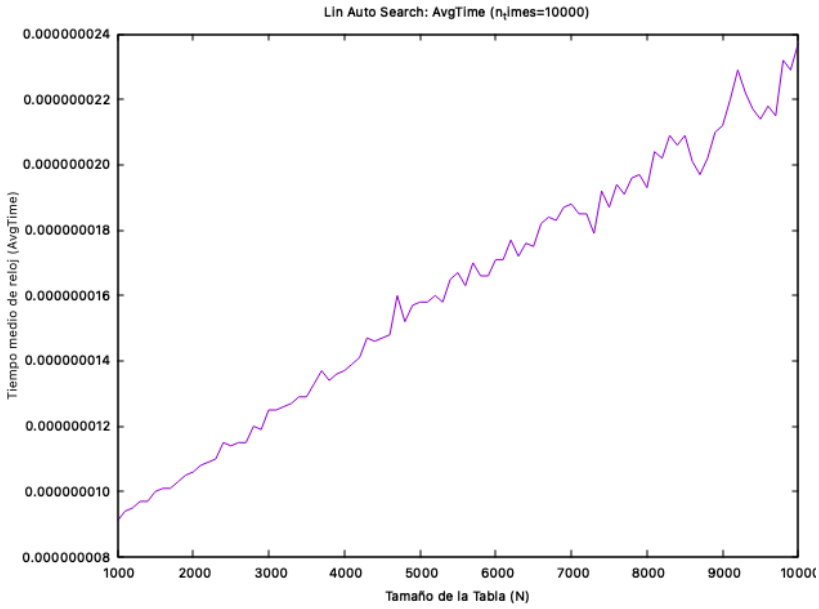
N_times = 100

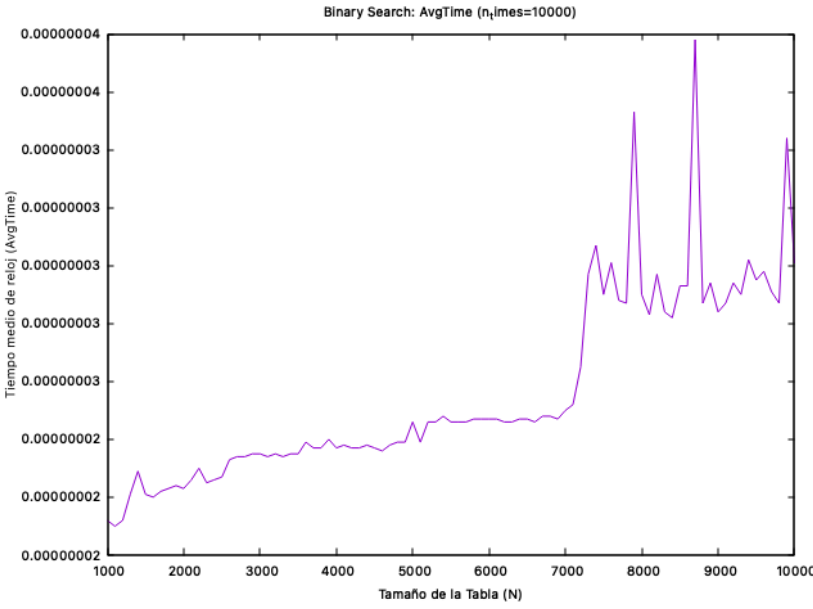




Ambas curvas se estabilizan. La Lineal Auto-organizada muestra una tendencia de crecimiento más pronunciada que la Binaria, debido al costo adicional del intercambio (reordenamiento) en cada acierto. La Búsqueda Binaria, al no tener costo de movimiento, muestra una tendencia de crecimiento muy lento (O(logN)) y es consistentemente más rápida.

N_times = 10000





Ambas curvas son mucho más suaves y estables. La Búsqueda Binaria mantiene un crecimiento mínimo que refleja algo su complejidad logarítmica (logN). La Lineal Auto-organizada también muestra una curva más suave pero claramente lineal, lo que evidencia que, aunque su promedio de comparaciones es O(1), el tiempo real de reloj se incrementa notablemente debido al costo de memoria y reordenamiento, siendo la Binaria la opción más eficiente en tiempo.

6. Respuesta a las preguntas teóricas.

6.1 ¿Cuál es la operación básica de bin search, lin search y lin auto search?

La operación básica tomada para todos los algoritmos fue la comparación de claves, ya que esta operación caracteriza muy bien los algoritmos de búsqueda, ya que depende de en qué posición se encuentre la clave en la tabla se necesitarán un número de comparaciones distintas que son proporcionales al tiempo de ejecución del algoritmo.

6.2 Dar tiempos de ejecución en función del tamaño de entrada n para el caso peor WSS (n) y el caso mejor BSS (n) de bin search y lin search. Utilizar la notación asinstótica (O,Θ,o,Ω,etc) siempre que se pueda.

Para la búsqueda lineal, el rendimiento depende directamente de la posición del elemento en la tabla desordenada. El caso mejor, $B_{lineal}(n)$, ocurre cuando la clave buscada se encuentra en la primera posición de la tabla, lo que permite terminar el algoritmo tras una única comparación, resultando en una complejidad asintótica de *O(1)*. Por el contrario, el caso peor $W_{lineal}(n)$ se produce cuando el elemento se encuentra en la última posición posible, de modo que el algoritmo debe recorrer y comparar los *n* elementos, resultando en una complejidad lineal de *O(n)*.

En el caso de la búsqueda binaria, que opera sobre tablas ordenadas dividiendo el espacio de búsqueda a la mitad, el comportamiento es más eficiente. Su caso mejor $B_{binaria}(n)$ se da cuando el elemento buscado coincide con la posición central de la tabla en la primera iteración, teniendo también una complejidad constante de *O(1)*. Sin embargo, su caso peor $W_{binaria}(n)$ ocurre cuando el elemento no está o la búsqueda se prolonga hasta reducir el subarreglo a un solo elemento; dado que el tamaño del problema se reduce a la mitad en cada paso (*n, n/2, n/4...*), el número de operaciones crece logarítmicamente, resultando en una complejidad de *O(log n)*.

6.3 Cuando se utiliza lin_auto_search y la distribución no uniforme dada ¿Cómo varía la posición de los elementos de la lista de claves según se van realizando más búsquedas?

El algoritmo intercambia cada elemento encontrado con su predecesor. En una distribución no uniforme, las claves más frecuentes son consultadas repetidamente, lo que provoca que escalen posiciones gradualmente con cada búsqueda hasta situarse en la cabecera de la lista. Como contrapartida, las claves menos solicitadas son desplazadas progresivamente hacia el final. Con el paso del tiempo, la estructura se auto-organiza quedando ordenada por frecuencia de uso descendente: los elementos más buscados quedan en los primeros índices, reduciendo drásticamente el tiempo medio necesario para localizarlos.

6.4 ¿Cuál es el orden de ejecución medio de lin auto search en función del tamaño de elementos en el diccionario n para el caso de claves con distribución no uniforme dado? Considerar que ya se ha realizado un elevado número de búsquedas y que la lista está en situación más o menos estable.

El orden de ejecución medio de la búsqueda lineal auto-organizada (lin_auto_search) para la distribución de claves no uniforme implementada en times.c (donde las claves pequeñas son mucho más probables) es *O(1)* (tiempo constante).

Esto ocurre porque, después de muchas búsquedas, las claves más frecuentes se mueven a las primeras posiciones de la lista, lo que garantiza que la mayoría de las búsquedas se realicen en un número constante y pequeño de comparaciones, independientemente del tamaño total del diccionario (N). No obstante, el peor caso se sigue manteniendo en *O(N),* ya que sigue, aunque sea muy baja, la probabilidad de que aparezca el último número de la tabla sigue estando.

6.5 Justifica lo más formalmente que puedas la corrección (o dicho de otra manera, el por qué busca bien) del algoritmo bin search.

La corrección del algoritmo se basa en que la tabla está ordenada y en la propiedad del Invariante de Bucle: *"Si la clave K existe en la tabla, necesariamente se encuentra dentro del rango actual de índices [F, L]"*.

El algoritmo funciona por reducción del espacio de búsqueda:

1. Al principio, el rango [*0, N-1*] cubre toda la tabla, por lo que si la clave existe, está dentro.
2. En cada iteración se compara la clave con el elemento central *T[i]*.
    - Si $K < T[i]$, la propiedad de orden garantiza que *K* no puede estar en la mitad derecha, por lo que es seguro descartarla y reducir el rango a [*F, i-1*].
    - Si $K > T[i]$, se garantiza que no está en la izquierda, reduciendo el rango a [*i+1, L*].
3. El rango se reduce en cada paso manteniendo siempre la garantía de que la clave no se ha "perdido". Si el algoritmo termina porque el rango se vacía ($F > L$) sin haber encontrado el elemento, podemos asegurar formalmente que la clave no existe en la tabla.

7. Conclusiones finales.

La práctica se centró en la implementación del TAD Diccionario y la medición experimental de la eficiencia de tres algoritmos de búsqueda: lineal, binaria y lineal auto-organizada. Los resultados obtenidos a partir de las Operaciones Básicas (OBs) y el Tiempo de Reloj confirmaron consistentemente las complejidades teóricas y permitieron evaluar el rendimiento de las estrategias.

Análisis de Complejidad Estática (Uniforme):

La primera fase, utilizando una distribución uniforme de claves, validó la diferencia fundamental de complejidad asintótica. La Búsqueda Lineal exhibió un crecimiento estrictamente lineal O(N) en el Número Medio de OBs, alcanzando aproximadamente 5,000 comparaciones para N=10000. Por otro lado, la Búsqueda Binaria mostró un crecimiento logarítmico (O(log N)). El análisis de la curva binaria experimental contra la función teórica f(x) = $log_2(x) - 1$ confirmó un ajuste casi perfecto, validando la implementación del algoritmo. Esta superioridad logarítmica resultó en tiempos de ejecución prácticamente despreciables para la Búsqueda Binaria en comparación con la Lineal.

Rendimiento Bajo Distribución No Uniforme:

El estudio de la Búsqueda Lineal Auto-organizada bajo la distribución potencial (no uniforme) demostró la efectividad de la heurística de reordenamiento. La Lineal Auto-organizada exhibió una rápida convergencia del Número Mínimo y Medio de OBs a O(1) (tiempo constante) a medida que n_times aumentó (100 y 10000). Esto ocurre porque el algoritmo de intercambio mueve consistentemente las claves más frecuentes a las primeras posiciones de la lista. Si bien el peor caso (Máximo OBs) se mantuvo en O(N), el rendimiento amortizado en el caso medio la convierte en una solución óptima para escenarios de acceso sesgado.

Impacto en el Tiempo de Reloj y la Memoria:

El análisis del Tiempo de Reloj Medio reveló que la Búsqueda Binaria es consistentemente más rápida que la Lineal Auto-organizada, ya que la Lineal Auto-organizada

incurre en una sobrecarga de tiempo significativa debido al costo del intercambio y reordenamiento en cada acierto. Además, las mediciones de tiempo de la Búsqueda Binaria mostraron caídas drásticas al cruzar ciertos tamaños de tabla (ej., alrededor de N=1000 en n_times=100. Estas variaciones se atribuyen a fenómenos de eficiencia de la caché de la CPU, que cambian la constante de tiempo oculta cuando la tabla deja de caber en la memoria caché.

En conclusión, la Búsqueda Binaria es la solución más robusta y eficiente en términos de rendimiento puro (O(log N)) y estabilidad temporal. Sin embargo, la Búsqueda Lineal Auto-organizada es la única que logra alcanzar una eficiencia de O(1) en tiempo medio para distribuciones de claves sesgadas, demostrando ser la estrategia adaptativa más eficiente para ese tipo de datos.