Análisis de Algoritmos 2022/2023

Práctica 3

David Brenchley Uriol, Javier San Andrés de Pedro, 1201.

Código	Gráficas	Memoria	Total

1. Introducción.

En esta práctica, vamos a construir el TAD diccionario, aunque, más propiamente dicho, lo que matemáticamente se describiría como un conjunto que contiene claves. Es decir, en la práctica, con lo único que vamos a trabajar es con las claves y no con los datos a los que dichas claves debieran de referenciar o, mejor dicho, indexar.

Por otro lado, también desarrollaremos funciones de búsqueda de claves en nuestro diccionario, así como nuevos medidores de los tiempos de ejecución para dichas funciones. Además, al igual que en la práctica anterior, referenciaremos de nuevo el trabajo anteriormente realizado, así como alguna que otra idea de alguno de los algoritmos de ordenación previamente desarrollados. Observaremos, además, que la eficiencia de dichos algoritmos dependerá de la frecuencia con la que aparezcan los datos y, en un intento de aprovecharse de dichas frecuencias, daremos con estructuras auto-organizadas.

2. Objetivos

2.1 TAD Diccionario

Implementación del TAD Diccionario con sus correspondientes funciones de creación, destrucción, adición de claves y búsqueda. Además, implementamos también distintos algoritmos de búsqueda.

2.2 Medición de tiempos de búsqueda

Elaboración de funciones especiales para la medición de tiempos de ejecución de algoritmos de búsqueda en diccionarios de diferentes tamaños. Además, también elaboraremos el rendimiento dependiendo de la frecuencia con la que se requieren las diferentes claves al diccionario.

3. Herramientas y metodología

La práctica se ha realizada en *Linux*, utilizando *Visual Studio* como herramienta para la programación del código correspondiente. Para la depuración de dicho código y el control de la reserva y fuga de memoria, se ha recurrido a *gcc* y *valgrind*. Finalmente, para la visualización de los resultados impresos por pantalla, se ha hecho uso se *sort* y *uniq*, además de *gnuplot* para la realización de gráficas.

En esta sección, comentaremos la metodología llevada a cabo para la resolución de los diferentes problemas y su correspondiente solución. Por otro lado, comentamos de manera breve las entradas y salidas de cada una de las rutinas; aunque para especificaciones más concretas, cada función viene bien documentada en los ficheros que contienen los códigos fuentes (por eso no consideramos necesario explayarnos en ello).

3.1 TAD Diccionario

En este apartado, simplemente hemos tenido que implementar ocho funciones (cinco correspondientes al TAD y tres relativas a la búsqueda). Comentaremos cada una de ellas por encima.

Dentro de las funciones del TAD tenemos la que se encarga de inicializar el diccionario: init_dictionary. Esta función se encarga de reservar memoria para una estructura de tipo diccionario, poner n_data (número de claves) a 0, asignarle un tamaño al diccionario y si las claves van a estar ordenadas o no. Además, se tiene que encargar de reservar memoria para el array que va a contener todas las claves que haya en el diccionario.

La siguiente función, que se encarga de destruir el diccionario, simplemente libera la memoria reservada para la tabla de claves y, seguidamente, eliminar la memoria reservada para la estructura diccionario en la función de inicialización.

En otro plano, tenemos la función que se encarga de insertar una única clave en el diccionario. Lo primero, se debe comprobar que, efectivamente, tenemos espacio para esa nueva clave. Una vez confirmamos que sí, debemos distinguir el caso en que la inserción se hace de manera ordenada del que no. En el caso primero, hemos tomado parte del algoritmo BubbleSort para insertar el elemento ya ordenado. Básicamente, lo que hacemos es insertar el elemento nuevo e ir intercambiándolo con todos aquellos que son mayores que él hasta que llegamos a uno que no lo es. En el caso en que no se inserta ordenadamente, simplemente se inserta al final.

También, hemos comprobado que efectivamente las claves se insertan ordenadas mediante un assert a la función ya diseñada que comprueba si un determinado array está ordenado. En este caso, le pasamos el campo table del diccionario (que es donde permanecen las claves).

Además, también se ha diseñado una función que inserta claves de manera masiva. Mediante un assert, controlamos que no se introduzcan más claves de las que el diccionario pudiere permitir insertar. A partir de ahí, lo único que hacemos es llamar tantas veces a la función de inserción como claves queramos insertar. En caso de ejecutar

el programa sin la bandera que activa los asserts e insertar más claves de las que el diccionario pudiere soportar, se insertarían tantas como espacio libre tuviere el diccionario.

Por último, tenemos la función que se encarga de buscar una clave en el diccionario. Simplemente se encarga de implementar una determinada función de búsqueda que se le pasa como argumento sobre el array que contiene las claves almacenadas.

Las posteriores funciones, corresponden a los diferentes algoritmos de búsqueda. La primera, la búsqueda lineal que se encarga de, empezando desde el principio, ir recorriendo elemento a elemento hasta que encuentra o no el elemento que desea. La segunda, es la búsqueda binaria que solo puede tiene sentido para diccionarios con claves ordenadas. Este algoritmo lo hemos implementado de acuerdo al siguiente pseudocódigo:

```
ind BBin(Tabla T, ind P, ind U, clave k)
    mientras P≤U :
        m=(P+U)/2;
        si T[m]==k :
            devolver m;
        else si k < T[m] :
            U=m-1;
        else :
            P=m+1;
    devolver error;
```

En ambas funciones de búsqueda, contabilizamos también el número de OBs, que en ambos casos hemos considerado que es la CDC.

Por último, cabe mencionar la última función lin_auto_search. Es exactamente igual que lin_search pero con la diferencia de que si encuentra el elemento, entonces lo intercambia por aquel que está justo delante de él (si tiene claro). La manera de implementarla es haciendo uso de la función anterior y es lin_search la que se encarga de contar el número de operaciones básicas.

Esta función es propia de una estructura auto-organizada ya que nos permite modificar la propia estructura de datos a medida que vamos interactuando con ella. Así pues, dependiendo de la frecuencia con que busquemos las claves, podremos conseguir mejores rendimientos a la hora de encontrarlas. Por otro lado, decir que hemos optado por aplicar la función que intercambia el elemento con el anterior ya que nos parece la más eficaz. No obstante, hay otras opciones como intercambiar el elemento por el primero o tratar de inferir la frecuencia de las claves solicitadas e ir organizando nuestra estructura de acuerdo a eso. Sin embargo, la primera no nos parece útil para el caso general ya que, aunque bien es cierto que la tendencia de las claves es a estar organizadas de mayor a menor frecuencia, también lo es que cualquier encuentro de clave modifica este orden constantemente. La segunda opción, sin embargo, es demasiado compleja.

3.2 Medición de tiempos de búsqueda

En este apartado, hemos tenido que adecuar las funciones de medición de la primera práctica a unas nuevas funciones específicas. Para ello, hemos tenido que implementar la función average_search_time que es el equivalente a la función average_sorting_times de la práctica 1, pero diseñada explícitamente para diccionarios.

Esta función se encarga de generar un diccionario de tamaño N y rellenarlo con N claves (mediante inserción masiva). Después, se encarga de reservar memoria para un array de tantos elementos como indique el campo de la estructura TIME_AA n_elems. En este caso, el número de elementos equivale al número de claves (N) por el número de veces que se busca cada clave (n_times). Además, debe rellenar este array llamando a una determinada función de generación de claves (esto nos va a servir para poder comparar los diferentes algoritmos de búsqueda con determinadas funciones). Una vez hecho esto, se tiene que encargar de completar tiempos medios, operaciones básicas medias, mínimas y máximas correspondientes de la búsqueda de N claves n_times veces en un determinado diccionario (que va a estar a SORTED) y con un determinado orden de búsqueda en las claves (también número). Se encargará, al igual que su equivalente en algoritmos de ordenación, de rellenar la estructura de time TIME_AA con sus respectivos campos.

Por otro lado, también hay que implementar la función generate_search_times que es la equivalente a la función generate_sorting_times de la práctica 1 y hace exactamente lo mismo; pero llamando, esta vez, a la función average_search_time. Básicamente, se encarga de generar mediciones de tantos diccionarios de tamaño variable entre un índice inferior y superior con un determinado incremento. Para las mediciones, llamará a la función anterior. Para imprimir los resultados en fichero, se servirá de la función save_time_table generada en la práctica 1.

4. Código fuente

4.1 TAD Diccionario

```
PDICT init_dictionary (int size, char order)
{
  PDICT dict;
  assert(size>0);
  assert(order==SORTED || order==NOT_SORTED);

  if ((dict=(PDICT) malloc(sizeof(DICT)))==NULL)
    return NULL;

  dict->size=size;
  dict->n_data=0;
  dict->order=order;

  if ((dict->table=(int*)malloc(size*sizeof(int)))==NULL)
  {
    free(dict);
    return NULL;
  }

  return dict;
}
```

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

```
int insert_dictionary(PDICT pdict, int key)
{
  int i;

  assert(pdict!=NULL);
  assert(key>=0);

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

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

```
  if (pdict->order==SORTED)
  {
    while (i>=1 && pdict->table[i-1]>key)
    {
      swap(&(pdict->table[i]), &(pdict->table[i-1]));
      i--;
    }

    assert(Array_is_Sorted(pdict->table, &(pdict->table[pdict->n_data-1]))==OK);
  }

  return OK;
}
```

```
int massive_insertion_dictionary (PDICT pdict,int *keys, int n_keys)
{
  int i;
  assert(pdict!=NULL);
  assert(keys!=NULL);
  assert(n_keys>0);
  assert(pdict->size-pdict->n_data-n_keys>=0);

  for (i=0; i<n_keys; i++)
    insert_dictionary(pdict, keys[i]);

  return OK;
}
```

Con tal de poder evaluar también el rendimiento cuando intentamos buscar claves que no aparecen en el diccionario. Hemos optado porque nuestra función de búsqueda no devuelva NOT_FOUND cuando no se encuentre la clave en la tabla (que es lo que el archivo exercise1.c parecería indicar que ha de devolver), sino que devuelva NOT_FOUND a través de ppos y devuelva el número de operaciones básicas que ha ejecutado. Con todo ello, vamos a presentar dos soluciones posibles: la primera de acuerdo a nuestras consideraciones y la consiguiente con respecto a lo que el archivo exercise1.c parecería indicar. Esto va a afectar explícitamente a la función de search_dictionary y la función de average_search_time.

Solución nuestra particular:

```
int search_dictionary(PDICT pdict, int key, int *ppos, pfunc_search
method)
{
  int ob;

  assert(pdict != NULL);
  assert(key > 0);
  assert(ppos != NULL);
  assert(method != NULL);

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

  return ob;
}
```

Solución según exercise1.c:

```
int search_dictionary(PDICT pdict, int key, int *ppos, pfunc_search method)
{
  int ob;

  assert(pdict != NULL);
  assert(key > 0);
  assert(ppos != NULL);
  assert(method != NULL);

  ob = method(pdict->table, 0, pdict->n_data - 1, key, ppos);
  if (*ppos == NOT_FOUND)
    return NOT_FOUND;

  return ob;
}
```

Ahora, vamos con las funciones de búsqueda

```
int bin_search(int *table, int F, int L, int key, int *ppos)
{
  int M, ob;

  assert(table != NULL);
  assert(F >= 0);
  assert(L >= F);
  assert(key >= 0);
  assert(ppos != NULL);

  ob = 0;
  while (F <= L)
  {
    M = (F + L) / 2;

    if (key == table[M])
    {
      *ppos = M;
      return ob+1;
    }
    else if (key < table[M])
      L = M - 1;
    else
      F = M + 1;

    ob++;
  }

  *ppos=NOT_FOUND;

  return ob;
}
```

```c
int lin_search(int *table, int F, int L, int key, int *ppos)
{
  int i, ob;

  assert(table != NULL);
  assert(F >= 0);
  assert(L >= F);
  assert(key >= 0);
  assert(ppos != NULL);

  i = F;
  ob = 0;
  while (i <= L)
  {
    ob++;
    if (table[i] == key)
      break;

    i++;
  }

  if (i > L)
    *ppos=NOT_FOUND;
  else
    *ppos = i;

  return ob;
}
```

```c
int lin_auto_search(int *table, int F, int L, int key, int *ppos)
{
  int ob;

  assert(table != NULL);
  assert(F >= 0);
  assert(L >= F);
  assert(key >= 0);
  assert(ppos != NULL);

  ob = lin_search(table, F, L, key, ppos);

  if (*ppos != NOT_FOUND && *ppos != F)
  {
    swap(&table[*ppos], &table[*ppos - 1]);
    (*ppos)--;
  }

  return ob;
}
```

4.2 Medición de tiempos de búsqueda

A esta función también afectan los cambios. Solución conforme a nuestras consideraciones:

```
short average_search_time(pfunc_search metodo, pfunc_key_generator generator, char
order, int N, int n_times, PTIME_AA ptime)
{
  int *keys, *dictkeys, i, pos, temp;
  double ini, fin;
  PDICT dict;

  assert(metodo != NULL);
  assert(generator != NULL);
  assert(ptime != NULL);
  assert((int)order == SORTED || (int)order == NOT_SORTED);
  assert(N > 0);
  assert(n_times > 0);

  if ((dict = init_dictionary(N, order)) == NULL)
    return ERR;

  if ((dictkeys = generate_perm(N)) == NULL)
  {
    free_dictionary(dict);
    return ERR;
  }

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

  ptime->N = N;
  ptime->n_elems = N * n_times;
  ptime->average_ob = 0;
  ptime->max_ob = 0;
  ptime->min_ob = INT_MAX;

  if ((keys = (int *)malloc(ptime->n_elems * sizeof(int))) == NULL)
  {
    free(dictkeys);
    free_dictionary(dict);
    return ERR;
  }

  generator(keys, ptime->n_elems, N); /*N reprsents the max key in this function*/
```

```
  ini = clock();

  for (i = 0; i < ptime->n_elems; i++)
  {
    temp=search_dictionary(dict, keys[i], &pos, metodo);

    if (temp < ptime->min_ob)
      ptime->min_ob = temp;

    if (temp > ptime->max_ob)
      ptime->max_ob = temp;

    ptime->average_ob += temp;
  }

  fin = clock();

  ptime->time = (double)(fin - ini) / CLOCKS_PER_SEC / ptime->n_elems;
  ptime->average_ob /= ptime->n_elems;


  free(dictkeys);
  free_dictionary(dict);
  free(keys);

  return OK;
}
```

En este caso, consideramos también las operaciones básicas que procedan de claves que no pertenezcan al diccionario. Además, no realizamos control de errores sobre search ya que no puede fallar, y si lo hiciera tenemos puestos los asserts por si acaso que abortarían la ejecución.

Solución de acuerdo a exercise1.c. En este caso, si la clave no está, no se tiene en cuenta de cara a contabilizar operaciones básicas.

```
short average_search_time(pfunc_search metodo, pfunc_key_generator generator, char
order, int N, int n_times, PTIME_AA ptime)
{
  int *keys, *dictkeys, i, pos, temp;
  double ini, fin;
  PDICT dict;

  assert(metodo != NULL);
  assert(generator != NULL);
  assert(ptime != NULL);
  assert((int)order == SORTED || (int)order == NOT_SORTED);
  assert(N > 0);
  assert(n_times > 0);

  if ((dict = init_dictionary(N, order)) == NULL)
    return ERR;

  if ((dictkeys = generate_perm(N)) == NULL)
  {
    free_dictionary(dict);
    return ERR;
  }

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

  ptime->N = N;
  ptime->n_elems = N * n_times;
  ptime->average_ob = 0;
  ptime->max_ob = 0;
  ptime->min_ob = INT_MAX;

  if ((keys = (int *)malloc(ptime->n_elems * sizeof(int))) == NULL)
  {
    free(dictkeys);
    free_dictionary(dict);
    return ERR;
  }

  generator(keys, ptime->n_elems, N); /*N reprsents the max key in this function*/
```

```
  ini = clock();

  for (i = 0; i < ptime->n_elems; i++)
  {
    temp=search_dictionary(dict, keys[i], &pos, metodo);

    if (pos!=NOT_FOUND)
    {
      if (temp < ptime->min_ob)
      ptime->min_ob = temp;

      if (temp > ptime->max_ob)
        ptime->max_ob = temp;

      ptime->average_ob += temp;
    }
  }

  fin = clock();

  ptime->time = (double)(fin - ini) / CLOCKS_PER_SEC / ptime->n_elems;
  ptime->average_ob /= ptime->n_elems;


  free(dictkeys);
  free_dictionary(dict);
  free(keys);

  return OK;
}
```

Sin embargo, estas consideraciones no afectan a esta función:

```
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)
{

  PTIME_AA pt;
  int i, j, n_sizes = ((num_max - num_min) / incr) + 1;

  assert(method != NULL);
  assert(generator != NULL);
  assert(file != NULL);
  assert(num_min > 0);
  assert(num_max >= num_min);
  assert(incr > 0);
  assert(n_times > 0);

  if ((pt = (PTIME_AA)malloc(n_sizes * sizeof(TIME_AA))) == NULL)
    return ERR;

  for (i = num_min, j = 0; i <= num_max; i += incr, j++)
  {

    if (average_search_time(method, generator, order, i, n_times, &pt[j]) == ERR)
    {
      free(pt);
      return ERR;
    }
  }

  if (save_time_table(file, pt, n_sizes) == ERR)
  {
    free(pt);
    return ERR;
  }

  free(pt);

  return OK;
}
```

Donde save_time_table es la función desarrollada en la práctica 1.

5. Resultados, Gráficas

5.1 TAD Diccionario

En este apartado, como ya se ha comentado, hemos tenido que desarrollar las funciones propias del TAD Diccionario más las de búsqueda cuyo rendimiento analizaremos posteriormente mediante gráficas. Al ejecutar el ejercicio1 con búsqueda lineal, obtenemos estos resultados:

```
javier@javier-VirtualBox:~/Downloads/Analisis-de-algoritmos-Practica_3$ make exercise1_test
Running exercise1
Pratice number 3, section 1
Done by: Your names
Group: Your group
Key 5 found in position 0 in 1 basic op.
javier@javier-VirtualBox:~/Downloads/Analisis-de-algoritmos-Practica_3$ make exercise1_test
Running exercise1
Pratice number 3, section 1
Done by: Your names
Group: Your group
Key 5 found in position 6 in 7 basic op.
```

Podría resultar más o menos sorprendente que diga que realizamos 7 operaciones básicas, pero que el elemento está en la posición 6. La respuesta a esto la podemos encontrar en la ejecución anterior y es que las posiciones en la tabla van del 0 al tamaño-1 de esa tabla.

Además, debido a lo que ya hemos expuesto en el apartado anterior, hemos modificado el archivo exercise1.c para que también nos indique las operaciones básicas realizadas en caso de que no encuentre la clave en la tabla.

```
nob = search_dictionary(pdict,key,&pos,lin_auto_search);

if(pos >= 0) {
  printf("Key %d found in position %d in %d basic op.\n",key,pos,nob);
} else if (pos==NOT_FOUND) {
  printf("Key %d not found in table. It took %d basic op.\n",key, nob);
} else {
  printf("Error when searching the key %d\n",key);
}
```

Ahora, se obtienen estos resultados:

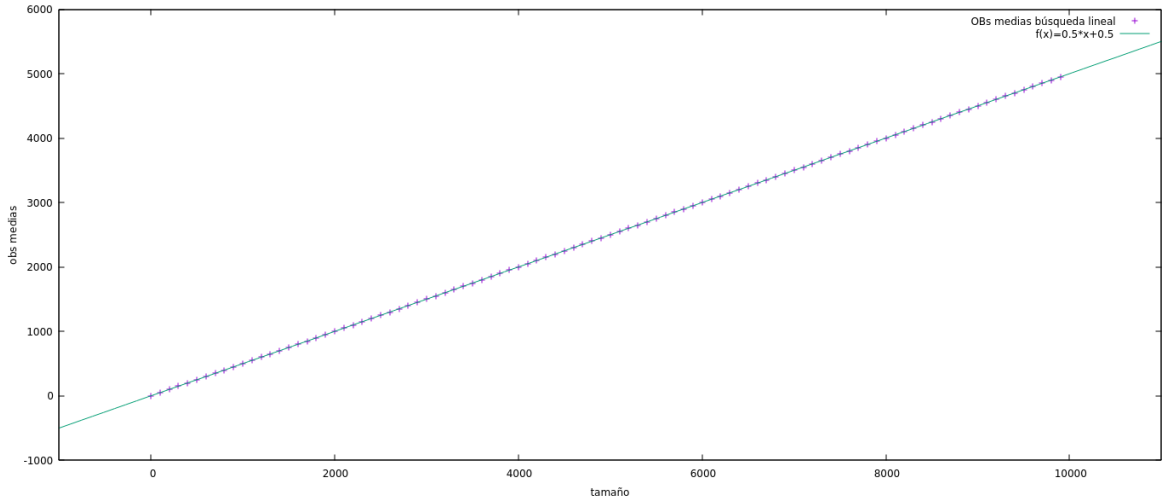
```
javier@javier-VirtualBox:~/Downloads/Analisis-de-algoritmos-Practica_3$ make exercise1_test
Running exercise1
Pratice number 3, section 1
Done by: Your names
Group: Your group
Key 57 not found in table. It took 10 basic op.
```

Haciendo lo propio con bin_search y lin_auto_search (aunque este último no tiene demasiado sentido), se comprueban también los resultados.

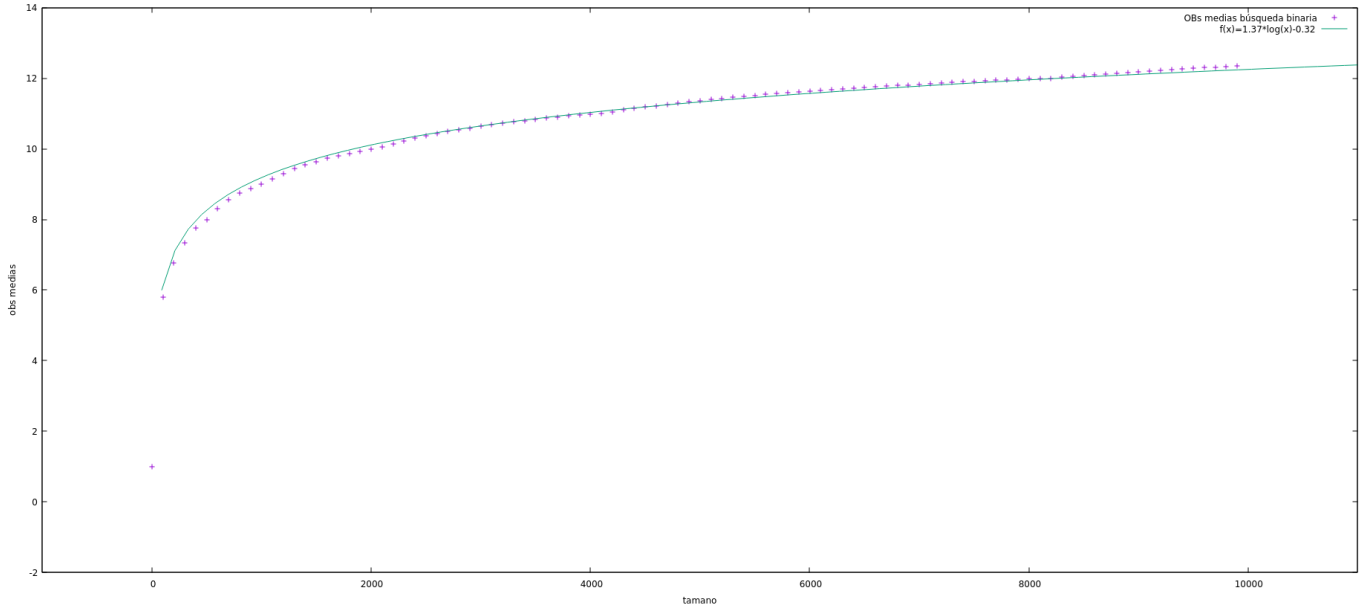
5.2 Medición de tiempos de búsqueda

La primera comparación de gráficas es entre la búsqueda lineal y búsqueda binaria. Vamos a comparar los tiempos medio de ejecución y el número de operaciones básicas medias. Empezaremos primero con el promedio de operaciones básicas.

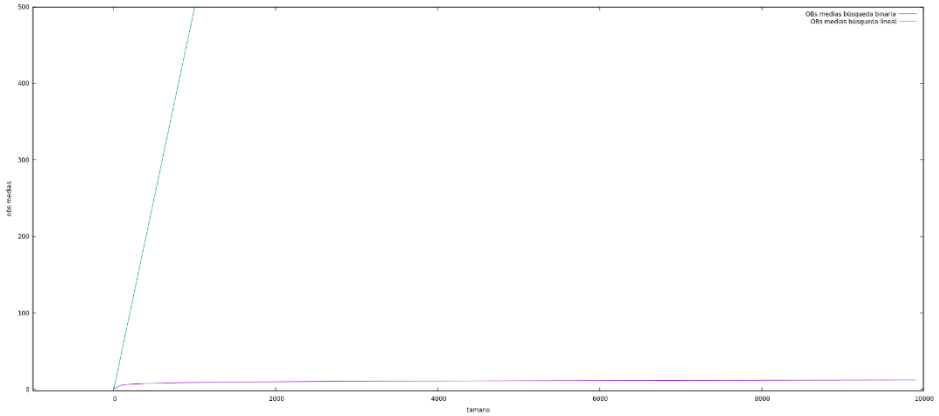
Esta es la gráfica correspondiente al número promedio de operaciones básicas de lin_search con tamaño máximo 10000, incr de 100 y n_times a 1:



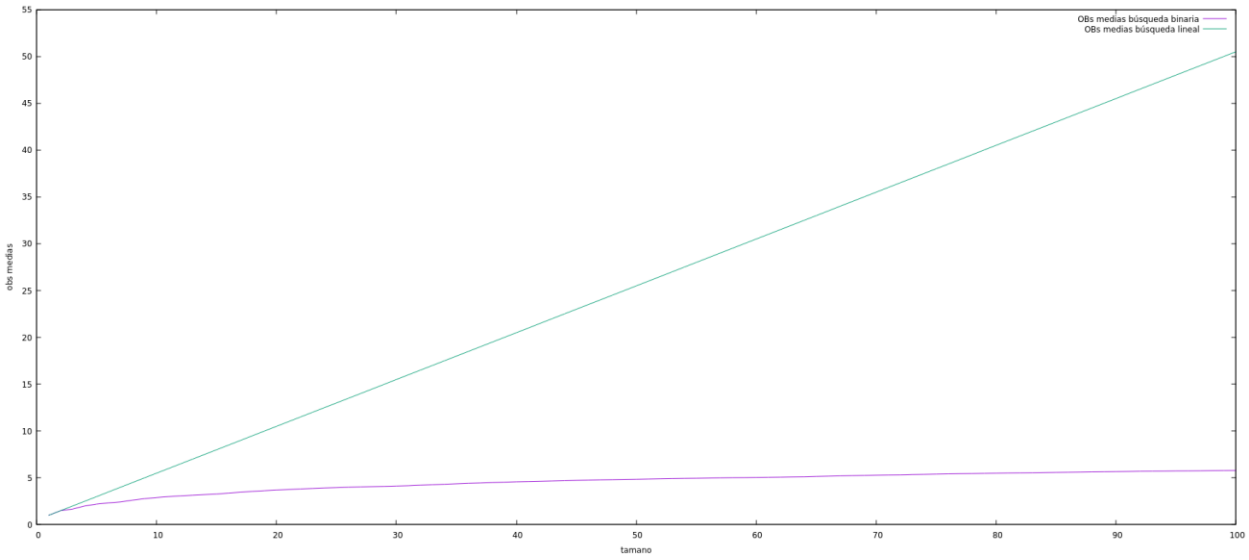
Por otro lado, tenemos las correspondientes a bin_search mismas condiciones:



Vemos ahora la comparativa entre las dos para tamaños grandes:

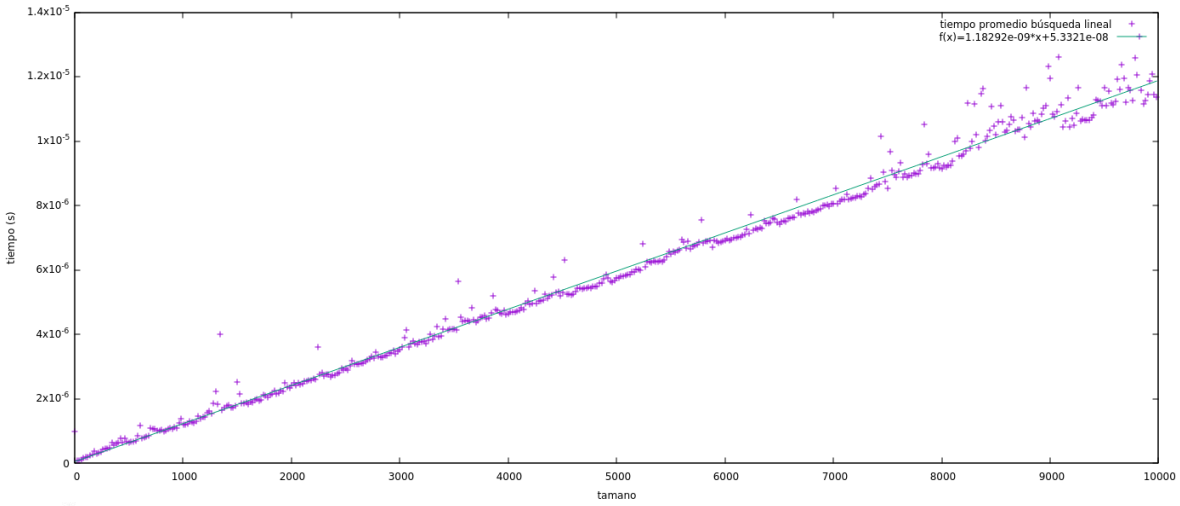


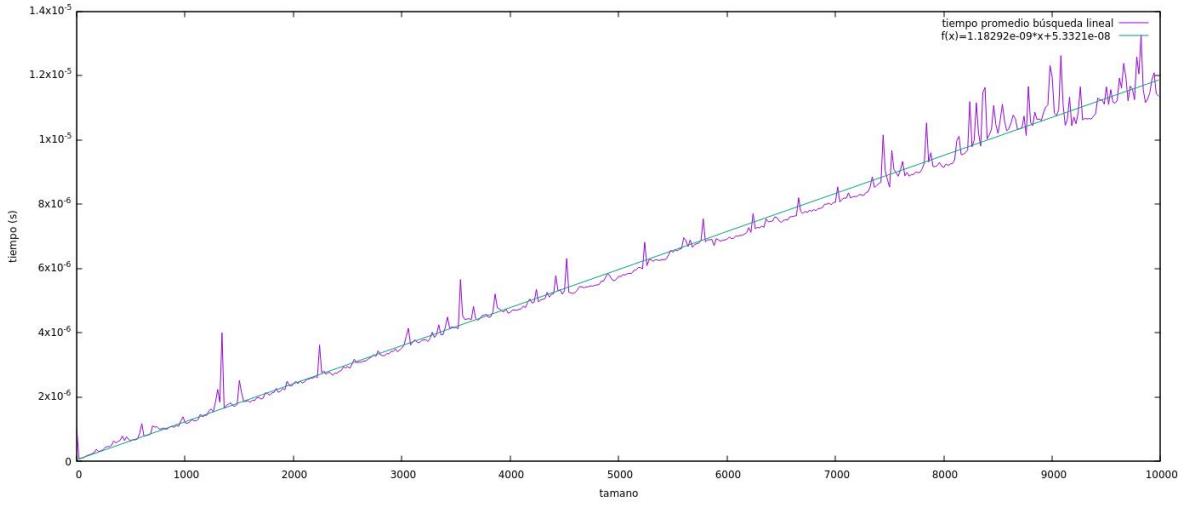
Como se puede apreciar, la comparativa no tiene mucho sentido porque bin_search parece casi lineal. Si reducimos un poco el tamaño obtenemos una gráfica donde evidencia la tendencia logarítmica de bin_search:



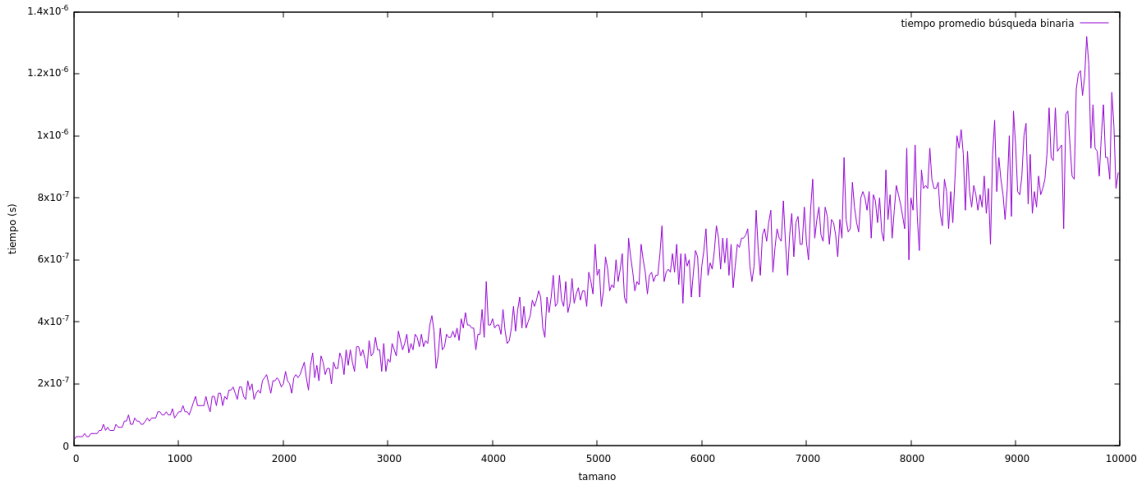
Obviamente y tal como se indica en los datos, se puede observar que la búsqueda binaria es mucho más eficiente que la búsqueda lineal. Solo para ponerlo en boga con datos numéricos. Para una tabla de 10000 la búsqueda lineal ejecutaría unas 5000.5 operaciones básicas de media, mientras que bin_search ejecutaría unas 14 como en su caso peor.

Ahora, vamos a analizar el tiempo promedio de ejecución. A continuación, tenemos el tiempo medio de ejecución de lin_search:

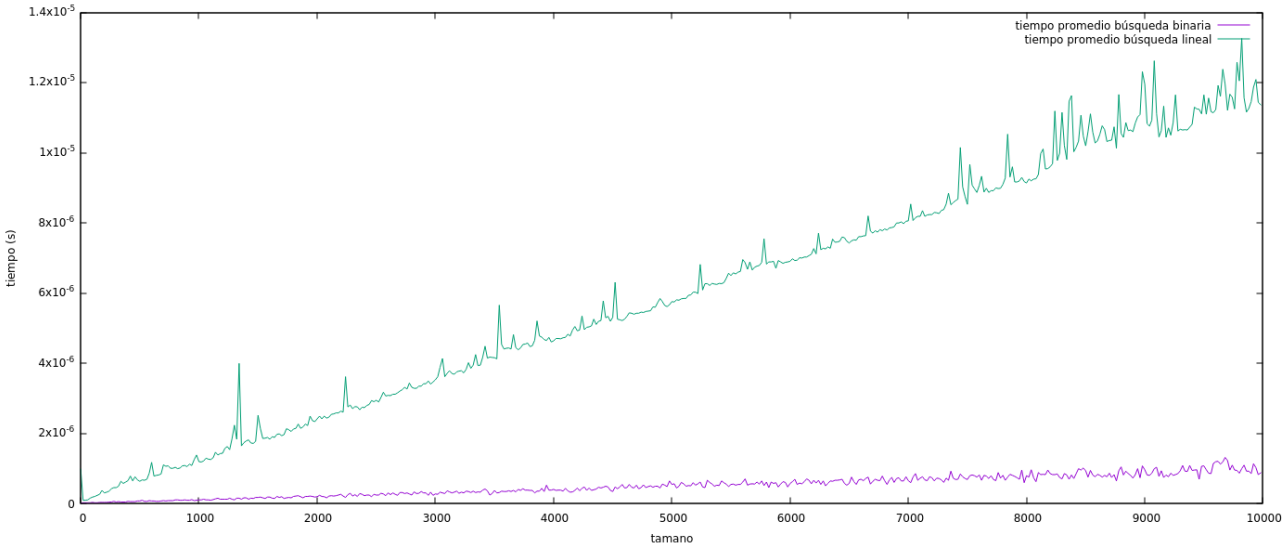




Por otra parte, tenemos la búsqueda binaria. En este caso, comentar que hemos necesitado incrementar el tiempo de n_times para que diera resultados más o menos “cabales”.



Se puede apreciar que ahora no tiene sentido ajustarla con una logarítmica ya que necesitaríamos mayor cantidad de datos para empezar a ver algún tipo de tendencia logarítmica. A continuación, exponemos la comparativa entre ambas:

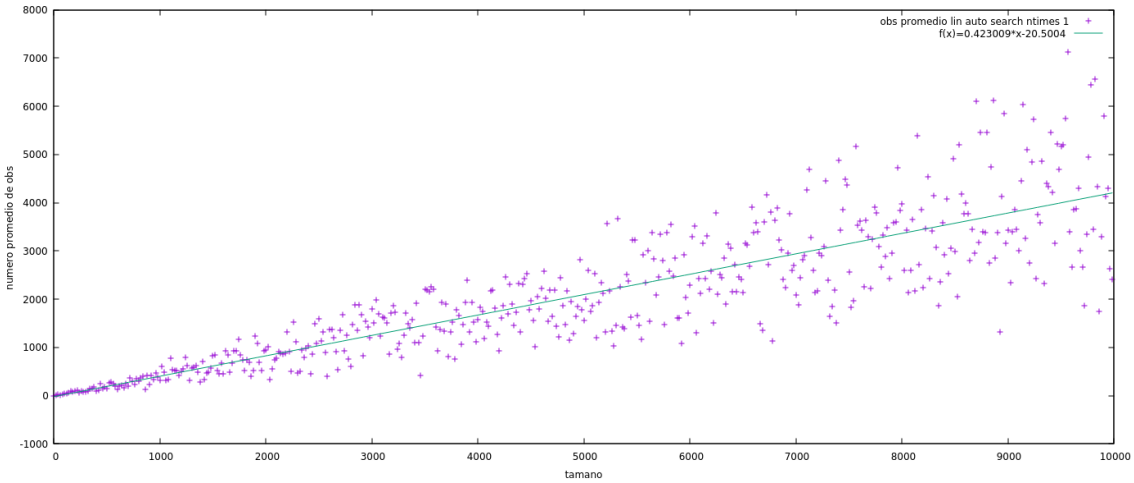


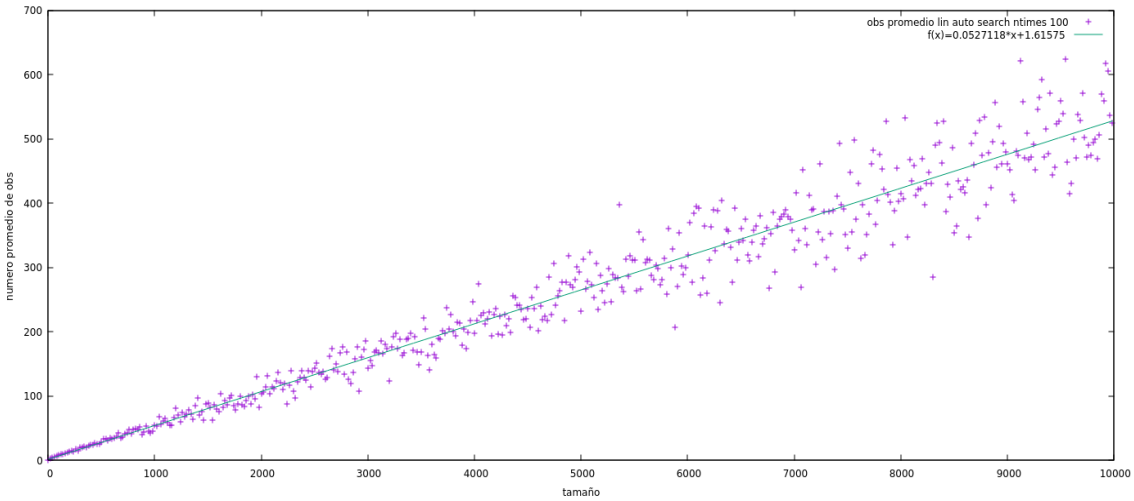
Ahora bien, podemos observar claramente también a nivel de tiempos de ejecución cómo la búsqueda binaria es mucho más ineficiente que la búsqueda lineal.

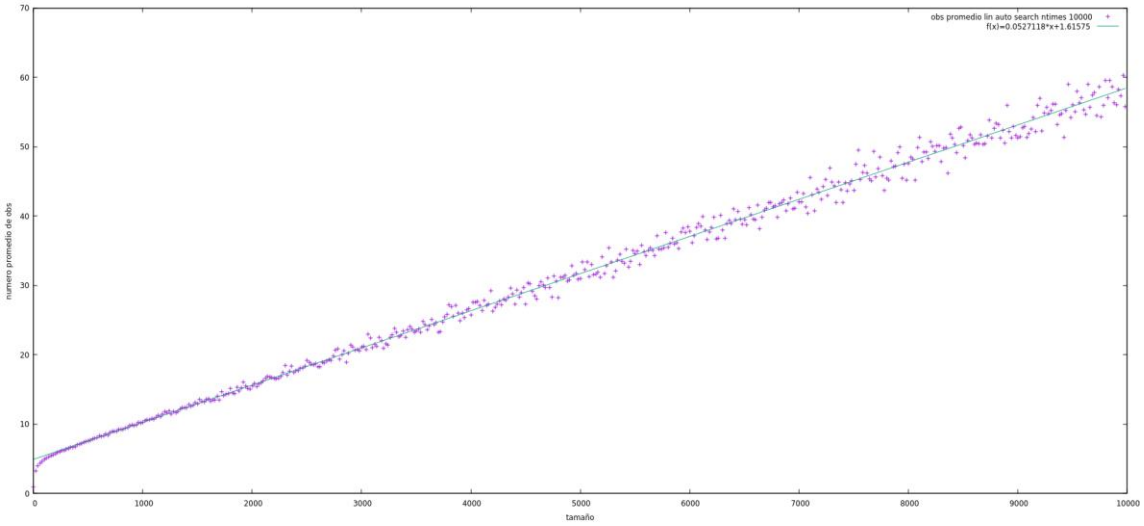
A continuación, empezaremos con las comparativas entre búsqueda binaria y búsqueda lineal. Empezaremos con comparar el número promedio de operaciones básicas:

Antes ya hemos expuesto las gráficas correspondientes a búsqueda binaria. Vamos a exponer ahora las de la búsqueda lineal para n_times=1, 100 y 10000.

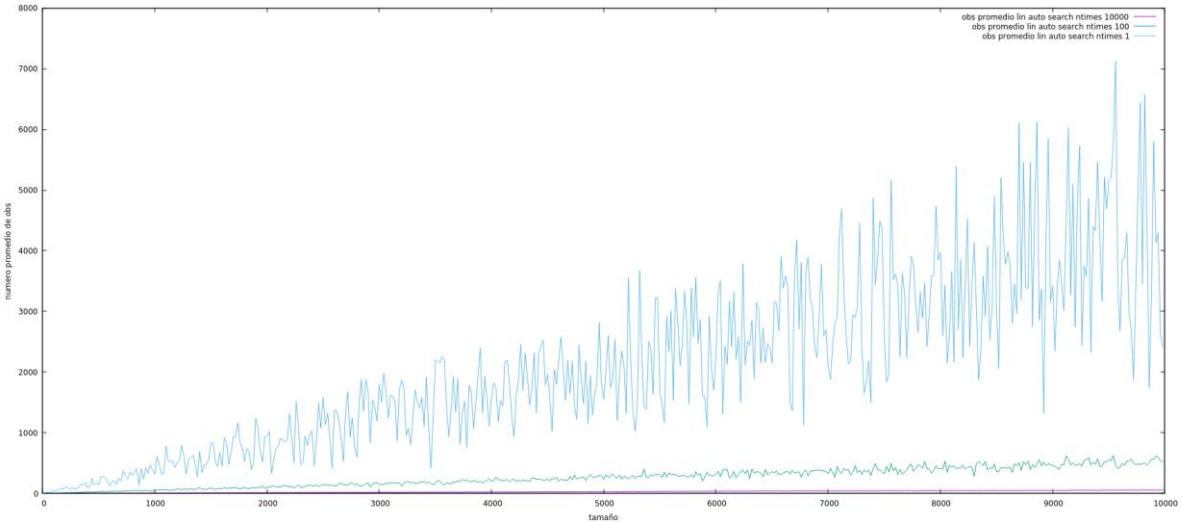
Vamos primero con las gráficas por separado y con su correspondiente fit:



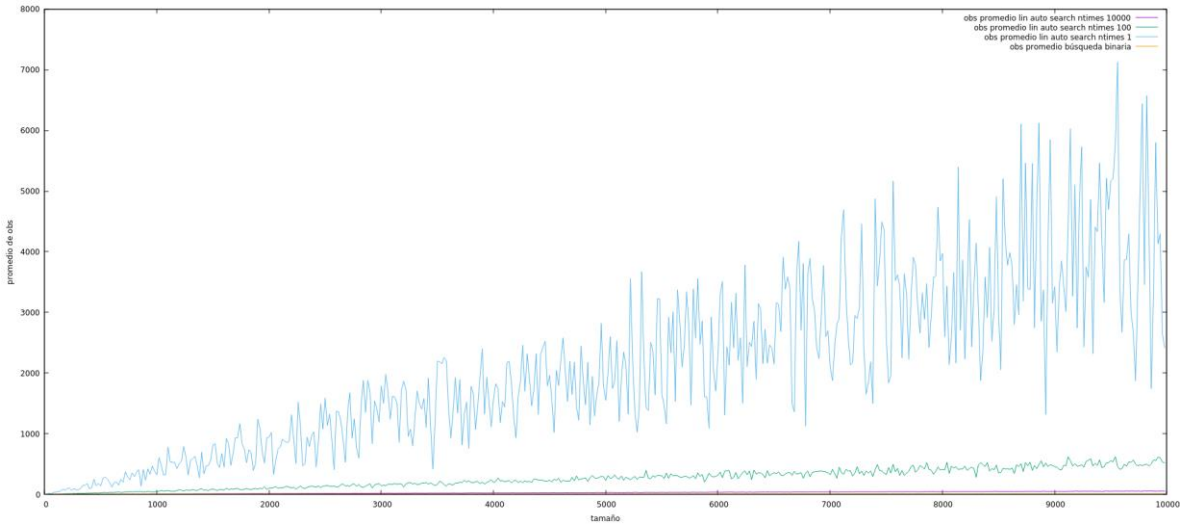


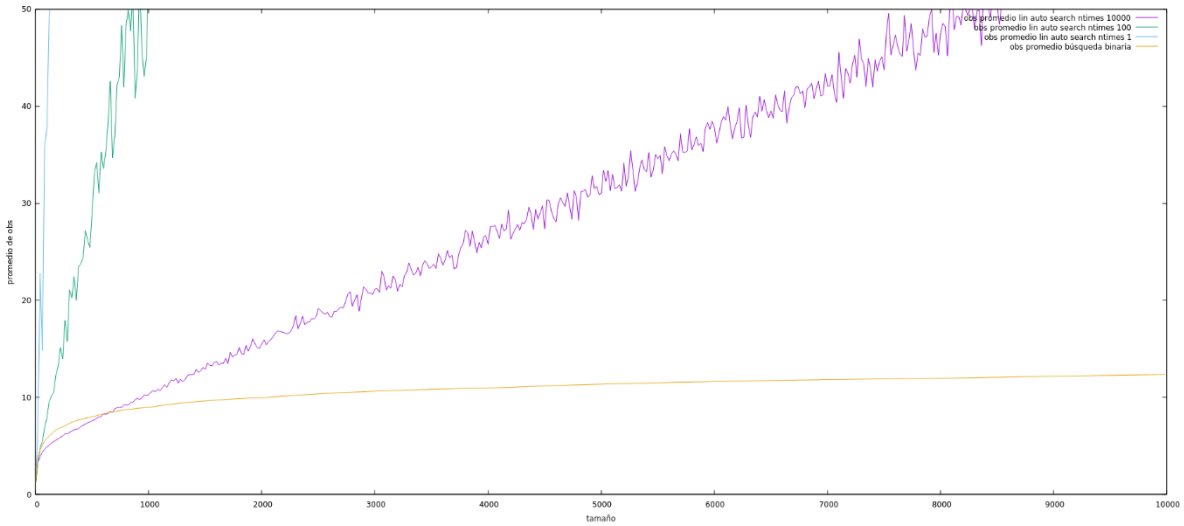


Estas dispersiones son debidas a que con n_times a 1, el número de operaciones básicas depende mucho de la posición de los primeros índices (que son los más probables a buscar). Podemos apreciar que los datos se van haciendo cada vez más uniformes a medida que aumentamos el número de veces.



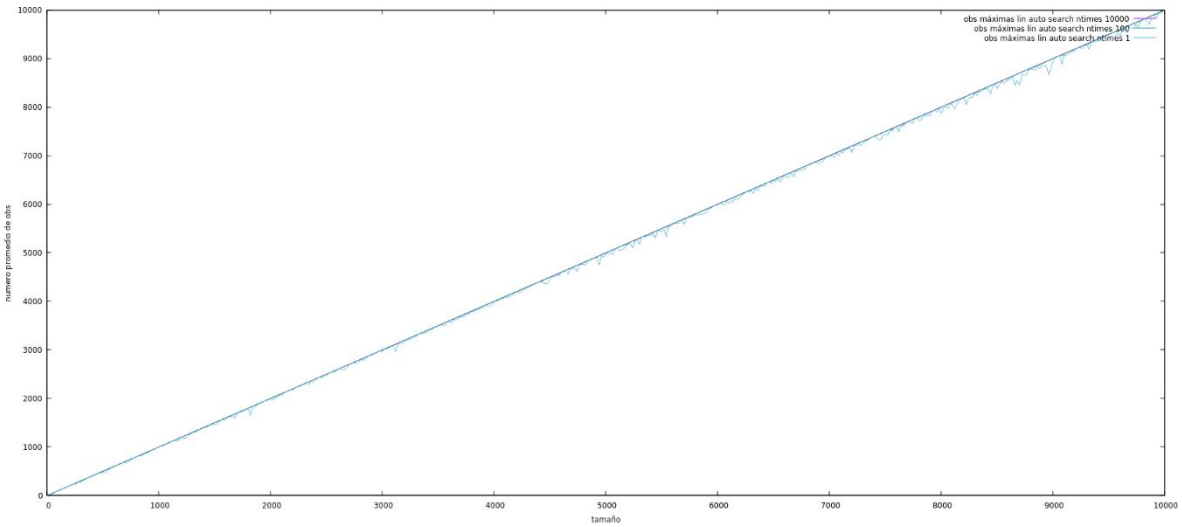
Ahora, veamos la comparativa con la búsqueda binaria:



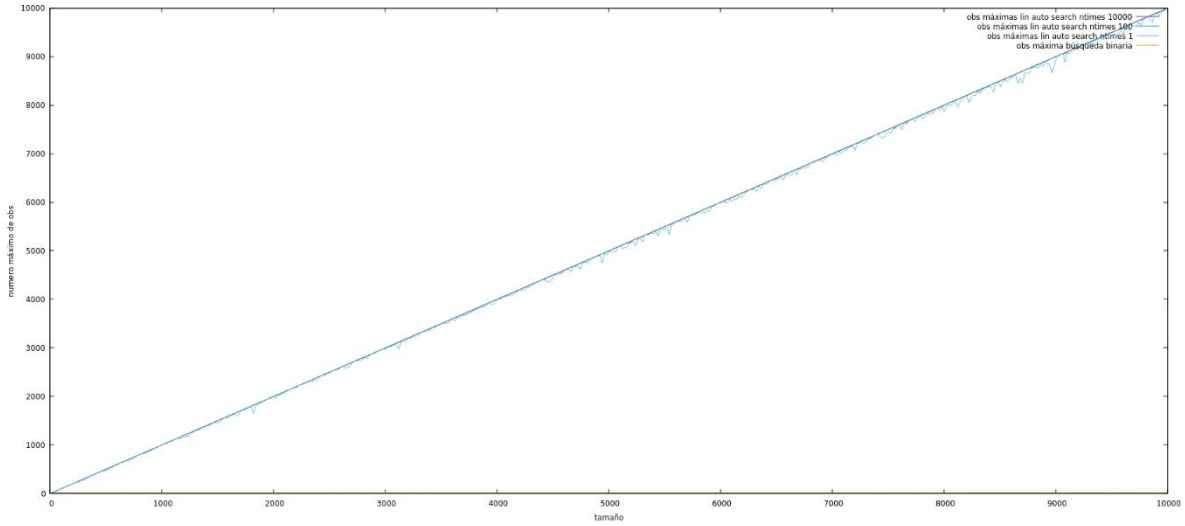


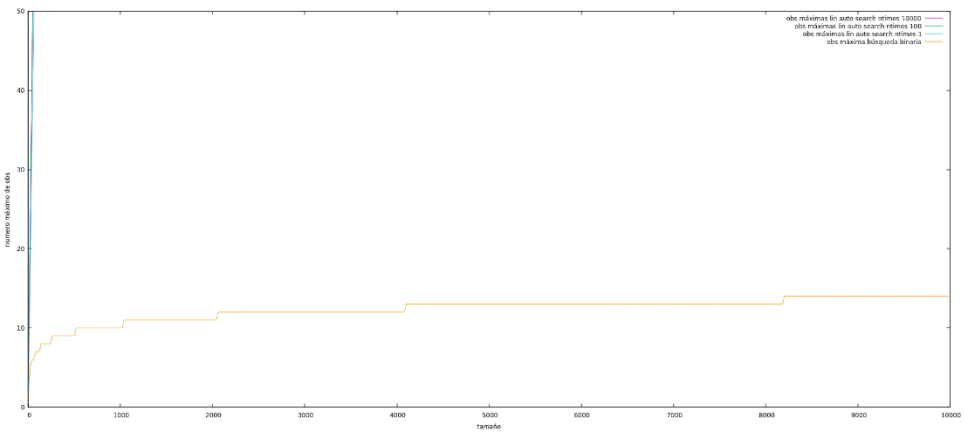
Aquí, se puede ver claramente cómo, a pesar de la distribución de los datos, se sigue viendo como bin_search tiene un rendimiento mejor. Esto se debe a que, hasta que lin_auto_search consigue esa estabilidad, se ejecutan bastantes operaciones básicas que le sacan fuera de cualquier competición con bin_search. Si partiéramos de tablas ordenadas, posiblemente nos diera un rendimiento mejor lin_auto_search (tal y como se expone en la pregunta 5).

Ahora, pasemos a comparar el número máximo de operaciones básicas entre ambos:



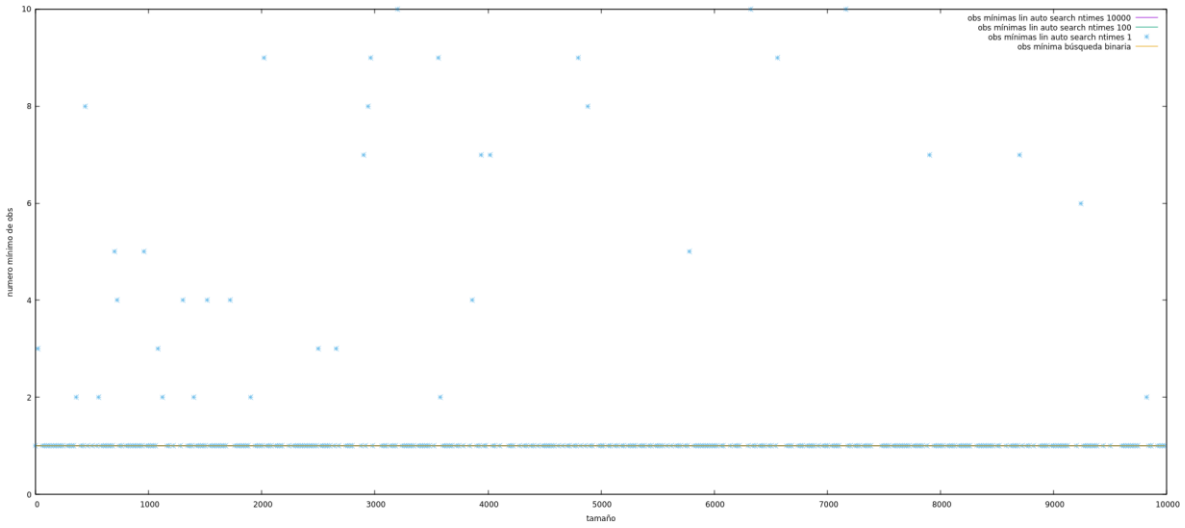
Estas son las gráficas sin tener en cuenta bin_search. Podemos observar como lin_auto_search con solo 1 repetición tiene a veces menos operaciones básicas. Esto se debe a que, al ejecutarse menos veces el generador, es más probable que no llegue a alcanzar todo el espectro de posibles valores.



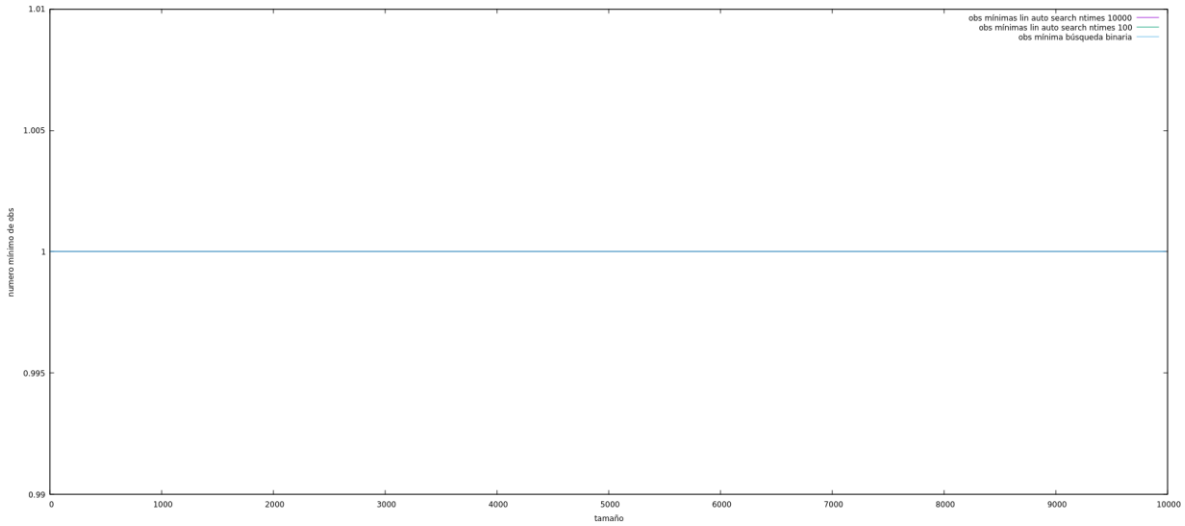


Como se puede ver, pasa igual que en el caso de lin_search. Tal y como evidenciamos aquella vez, el número máximo de operaciones básicas que puede ejecutar lin_auto_search sobre una tabla de 10000 elementos es 5000.5, mientras que bin_search realiza a lo sumo 14. Los escalones se deben a las potencias de 2, en donde el número máximo experimenta una variación de 1 unidad por la función techo de lg(X).

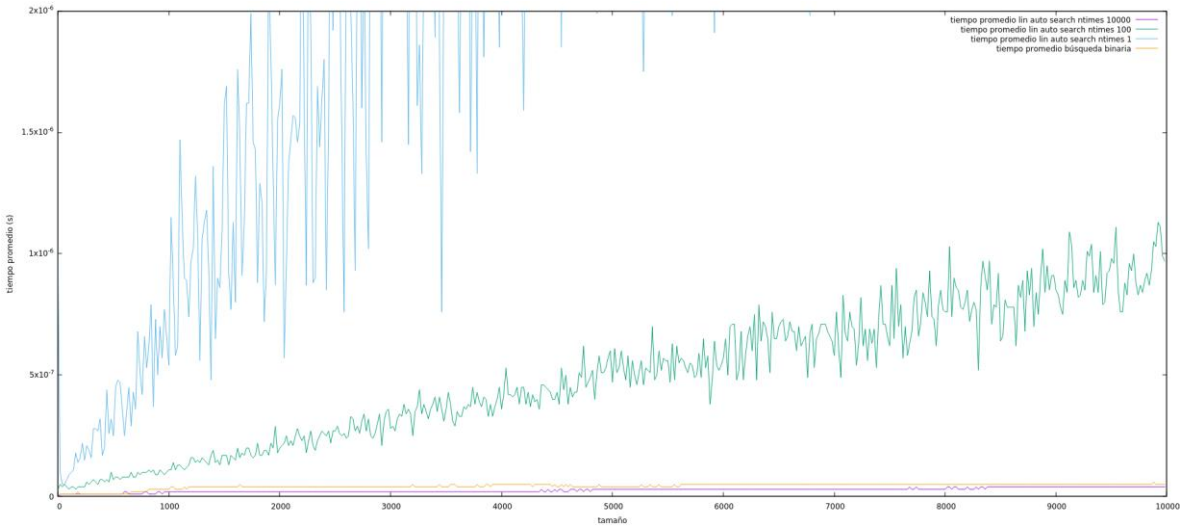
Comparemos ahora el número mínimo de operaciones básicas.

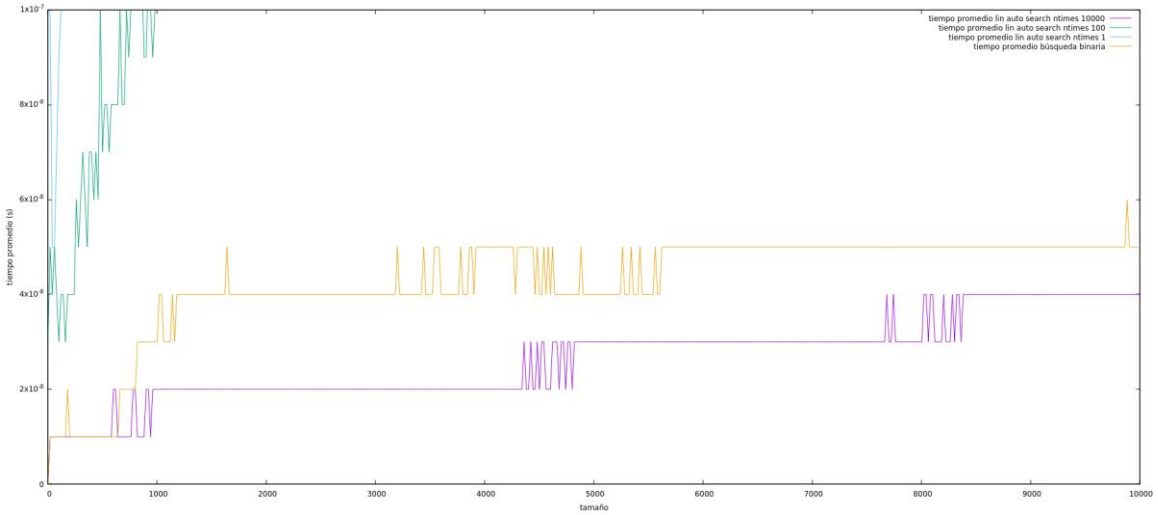


Lo que podemos apreciar es que tanto para los valores de n_times 10000, 100 y bin_search tienen como operación básica mínimo 1 operación; mientras que los datos varían casi caóticamente para n_times 1. La razón detrás de esto está en que dado el número de veces que ejecutamos el generador y la posición de los primeros elementos en la tabla, es muy probable que solo busquemos en elementos cuya distribución en la tabla nos haga requerir de mayor número de operaciones básicas. Si quitamos el n_times 1 de la gráfica se aprecia que cuanto más ejecutemos el generador más probable es barrer el espectro de todos los valores y preguntar por el elemento que esté en la primera posición:



Por último, vamos a analizar tiempo de ejecución promedio.





En este caso, sorprende ver que lin_auto_search con n_times a 10000 consigue un rendimiento en tiempo de ejecución mejor que el propio bin_search. Esto es debido a una cuestión probabilística y a que, tal y como mencionamos en el promedio de operaciones básicas entre ambos, una vez lin_auto_search alcanza una determinada estabilidad, consigue un rendimiento superior. Lo que debe explicar el resultado, que no se valora en el promedio de operaciones básicas, es que durante todo ese tiempo en que lin_auto_search no está ejecutándose en una tabla más o menos estable, la diferencia de tiempo con respecto a bin_search no es tan significativa como el número de operaciones básicas (estamos hablando de un tiempo de nanosegundos), por lo que luego consigue paliar ese arranque inicial dando ese resultado tan particular.

Es decir, dado que los valores de tiempo son pequeños, luego consigue marcar una diferencia. No ocurre lo mismo con el promedio de obs porque ahí la diferencia es significativa.

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 de los tres algoritmos es la CDC. Veamos sobre el propio código de las tres funciones de manera directa quién es esa clave de comparación.

```c
int bin_search(int *table, int F, int L, int key, int *ppos)
{
  int M, ob;

  assert(table != NULL);
  assert(F >= 0);
  assert(L >= F);
  assert(key >= 0);
  assert(ppos != NULL);

  ob = 0;
  while (F <= L)
  {
    M = (F + L) / 2;
    if (key == table[M])
    {
      *ppos = M;
      return ob+1;
    }
    else if (key < table[M])
      L = M - 1;
    else
      F = M + 1;

    ob++;
  }

  return NOT_FOUND;
}
```

Operación básica (CDC)

```c
int lin_search(int *table, int F, int L, int key, int *ppos)
{
  int i, ob;

  assert(table != NULL);
  assert(F >= 0);
  assert(L >= F);
  assert(key >= 0);
  assert(ppos != NULL);

  i = F;
  ob = 0;
  while (i <= L)
  {
    ob++;
    if (table[i] == key)
      break;

    i++;
  }

  if (i > L)
    return NOT_FOUND;


  else
    *ppos = i;

  return ob;
}
```

Operación básica (CDC)

(También es la operación básica de lin_auto_search ya que utiliza primero búsqueda lineal)

6.2 Dar tiempos de ejecución en función del tamaño de entrada n para el caso peor $W_{SS}(n)$ y el caso mejor $B_{SS}(n)$ de bin_search y lin_search. Utilizar notación asintótica (O, Θ, o, Ω , etc.) siempre que se pueda:

En el caso de lin_search, el caso peor es tener que recorrerse todos los elementos de la tabla. Luego se tiene, en primera instancia, para una tabla de tamaño N y cualquier permutación σ: $n_{LS}(\sigma, i) \leq N$. Por otro lado, sea $j = \sigma(N)$, el último elemento de la tabla de tamaño N (suponiendo que las posiciones varían de 1 a N y no de 0 a N-1). No es difícil ver que $n_{LS}(\sigma, j) \geq N$ (de hecho es igual). Sea k un elemento t.q. $\nexists i \; k = \sigma(i)$, es decir, un elemento que no pertenezca a la tabla. Entonces tenemos el mismo caso que con el índice j anterior. Con todo esto, se puede concluir que:

$$W_{LS}(N, k) = N$$

Con respecto al caso mejor, se tiene ahora que $n_{LS}(\sigma, i) \geq 1, \forall \sigma, i$. Sea ahora j el primer elemento de la tabla, $j = \sigma(1)$, entonces $n_{LS}(\sigma, j) \leq 1$, de hecho se da la igualdad. Por tanto:

$$B_{LS}(N, k) = 1$$

Pasamos ahora a hacer lo propio con bin_search. En el caso peor, tenemos que el algoritmo se ejecuta hasta que encuentra la clave o hasta que tenemos que F>L, en ese caso la clave no está en la tabla. Queremos evaluar $n_{BB}(T, F, L, k)$, no hemos añadido ppos porque no es relevante. Para simplificar, suponemos F=1, L=N, con N el tamaño de la tabla. Llamaremos a partir de ahora a $n_{BB}(T, 1, N, k)$ como $n_{BB}(T, N, k)$. Como la tabla está ordenada, no hace falta tener en cuenta ningún tipo de permutaciones.

$$n_{BB}(T, N, k) = \begin{cases} n_{BB}\left(T, \left\lfloor \frac{N}{2} \right\rfloor, k\right) + 1 \; si \; k < T\left(\left\lfloor \frac{N}{2} \right\rfloor\right) \\ n_{BB}\left(T, \left\lfloor \frac{N}{2} \right\rfloor, k\right) + 1 \; si \; k < T\left(\left\lfloor \frac{N}{2} \right\rfloor\right) \\ 1 \qquad\qquad\qquad\qquad si \; k = T\left(\left\lfloor \frac{N}{2} \right\rfloor\right) \end{cases}$$

Con todo ello, se puede observar que el caso peor corresponde a no encontrar la clave o a encontrarla cuando el tamaño de la tabla es 1. Esto es debido a que así maximizamos el número de llamadas recursivas y, consecuentemente, el número de veces que se ejecuta la operación básica. Como todo el rato estamos partiendo la tabla en 2 mitades se puede apreciar que el algoritmo se acabará cuando el tamaño sea 1.

Por tanto, $n_{BB}(T, N, k) \leq \lceil \lg(N) \rceil$, donde lg es el logaritmo en base 2. Supongamos, un $N=2^m$, una potencia de 2, entonces se tendría que:

$$n_{BB}(T, N, k) = \begin{cases} n_{BB}\left(T, \frac{N}{2}, k\right) + 1 \; si \; k \neq T\left(\frac{N}{2}\right) \\ 1 \qquad\qquad\qquad\qquad si \; k = T\left(\frac{N}{2}\right) \end{cases}$$

$$n_{BB}(T,N,k) \leq n_{BB}\left(T,\frac{N}{2},k\right)+1$$

Desarrollando por inducción y sabiendo que $n_{BB}(T,1,k)=0$, se llega a que $n_{BB}(T,1,k) \leq m = \lg(N)$.Por otro lado, para un N general, se puede demostrar inductivamente que $n_{BB}(T,N,k) \leq \lceil \lg(N) \rceil$. De hecho, para el caso de no encontrar la clave se da la desigualdad reflexiva y, por ende, se puede concluir que:

$$W_{BB}(N,k) = O(\lg(N))$$

Para el caso mejor, basta ver por la desigualdad igualdad inicial que $n_{BB}(T,N,k) \geq 1$. Sea j= $T\left(\frac{N}{2}\right)$. Entonces $n_{BB}(T,N,j) \leq 1$ (de hecho, se da la igualdad). Por tanto, se puede afirmar que:

$$B_{BB}(N,k) = 1$$

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?

Lo primero, tal y como desarrollaremos en la siguiente pregunta, se tiene que la probabilidad de que un índice i aparezca es, si max es el máximo que se le ha indicado al generador potencial:

$$p(i) = \begin{cases} \frac{1}{3}+\frac{1}{max} & si\ i=1 \\ \frac{1}{i^2-\frac{1}{4}} & si\ 2 \leq i \leq max-1 \\ \frac{2}{2i-1}-\frac{1}{max} & si\ i=max \end{cases}$$

Por tanto, se puede ver de manera muy clara que la función es estrictamente decreciente, lo cual hace que sea menos probable que aparezcan índices cercanos a max.

Por ello, con el tiempo, lo más probable es que lin_auto_search ordene por probabilidad los elementos de la tabla. En este caso, eso es equivalente a ordenar la propia tabla. Esto es debido a que el 1 que es el más probable a salir, va a salir muchas más veces y será llevado a la primera posición con el tiempo. De igual forma, podemos realizar un argumento similar para todos los i.

Obviamente, por cada generación y búsqueda de esa clave estamos sujetos a perder el orden; pero la tendencia es a recuperarlo con sucesivas búsquedas.

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.

Para esta pregunta, necesitamos primero calcular la función de probabilidad de aparición de los determinados i a buscar dado el generador de claves:

```
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;
}
```

Como keys[i] es siempre un entero, tenemos realmente la parte entera de lo de la derecha del igual. Sea n el número entero que queremos obtener de la fórmula y sea α la división de rand()/RAND_MAX Entonces:

$$n = \left\lfloor \frac{1}{2} + \frac{max}{1+\alpha \cdot max} \right\rfloor \longrightarrow n \leq \frac{1}{2} + \frac{max}{1+\alpha \cdot max} < n+1$$

Resolviendo las dos desigualdades y despejando α se tiene que:

$$\frac{2}{2n+1} - \frac{1}{max} < \alpha \leq \frac{2}{2n-1} - \frac{1}{max}$$

Además, dado que $rand() \in [0, RAND\_MAX]$, se tiene que $0 \leq \alpha \leq 1$. Por ende, habrá que tener en cuenta cuando los elementos que están justo a izquierda y derecha de la desigualdad son inferiores a 0 o superiores a 1 respectivamente.

Resolviendo las respectivas desigualdades, tenemos que $\frac{2}{2n-1} - \frac{1}{max} > 1, para\ n \leq 1$. Además, como max>1, se tiene que $\frac{2}{2n+1} - \frac{1}{max} > 1\ para\ n \leq 0$. Por tanto, eso demuestra que no se pueden generar valores menores que 1.

Por otro lado, queda ver cuando $\frac{2}{2n+1} - \frac{1}{max} < 0$. La solución es justo para $n \geq max$. Además, como max>1, se tiene que $\frac{2}{2n-1} - \frac{1}{max} < 0\ para\ n > max$. Lo cual demuestra que no se pueden generar valores mayores que max.

Por tanto, asumiendo equiprobabilidad para encontrar rand() y un RAND_MAX lo suficiente grande. Podemos decir que:

$$p(\alpha) \approx \frac{2}{2n-1} - \frac{1}{max} - \frac{2}{2n+1} + \frac{1}{max} = \frac{1}{n^2 - \frac{1}{4}} = p(n)$$

Ahora bien, hay que tener cuidado con n=1 y n=max por lo comentado anteriormente. En esos casos:

$$p(\alpha) \approx 1 - \frac{2}{3} + \frac{1}{max} = \frac{1}{3} + \frac{1}{max} = p(1)$$

$$p(\alpha) \approx \frac{2}{2max - 1} - \frac{1}{max} - 0 = \frac{2}{2max - 1} - \frac{1}{max} = p(max)$$

En definitiva, podemos afirmar que:

$$p(i) = \begin{cases} \frac{1}{3} + \frac{1}{max} & si\ i = 1 \\ \frac{1}{i^2 - \frac{1}{4}} & si\ 2 \leq i \leq max - 1 \\ \frac{2}{2i - 1} - \frac{1}{max} & si\ i = max \end{cases}$$

No obstante, estos resultados no son 100% exactos porque rand() es siempre un número entero. La probabilidad exacta tiene una fórmula más fea de este estilo:

$$p(\alpha) \approx \frac{\left\lfloor \left(\frac{2}{2n - 1} - \frac{1}{max} \right) \cdot RAND\_MAX \right\rfloor - \left\lfloor \left(\frac{2}{2n + 1} + \frac{1}{max} \right) \cdot RAND\_MAX \right\rfloor}{RAND\_MAX} = p(n)$$

Aquí también habría que tener en cuenta por separado el caso n=1 y n=max. No obstante, nosotros nos contentaremos con la fórmula anterior.

Ahora bien, nos están pidiendo que discutamos el caso medio cuando la tabla está en un estado más o menos estable. Tal y como hemos desarrollado en la pregunta anterior, eso significa que la tabla está ordenada. Como solo buscamos en elementos que estén en la tabla y en una tabla de 1 a N:

$$A_{LAS}(N, k) = \sum_{i=1}^{N} p(i) \cdot n_{LAS}(N, i) = p(1) + \sum_{i=2}^{N-1} p(i) \cdot i + p(N) \cdot N$$

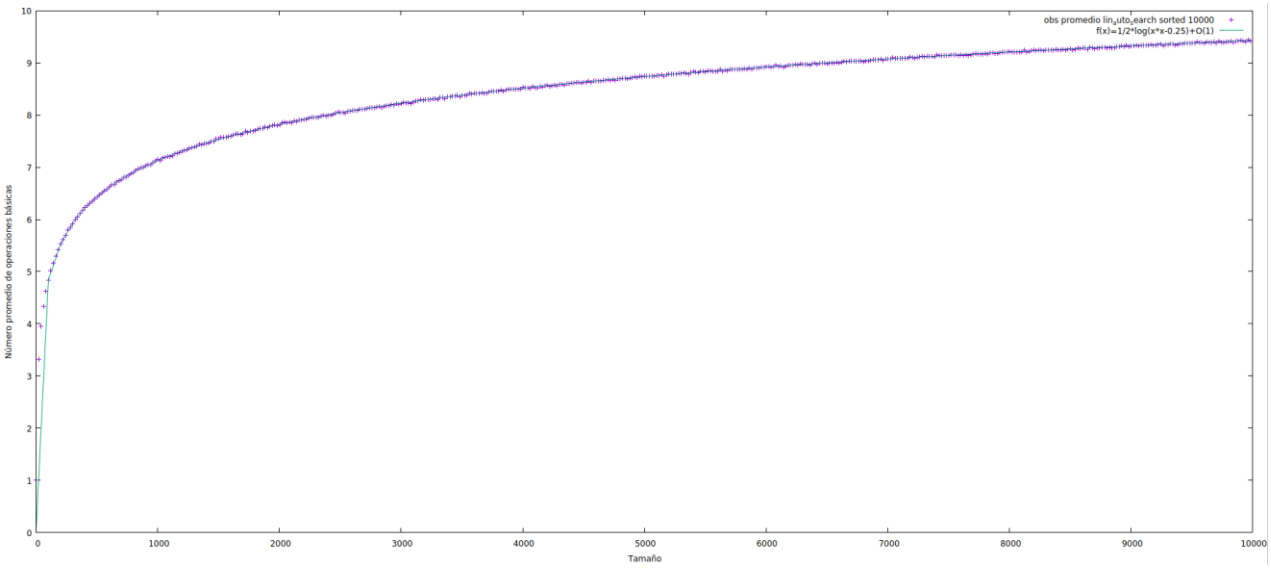
En nuestro caso, además, max=N para todos los tamaños. Por tanto:

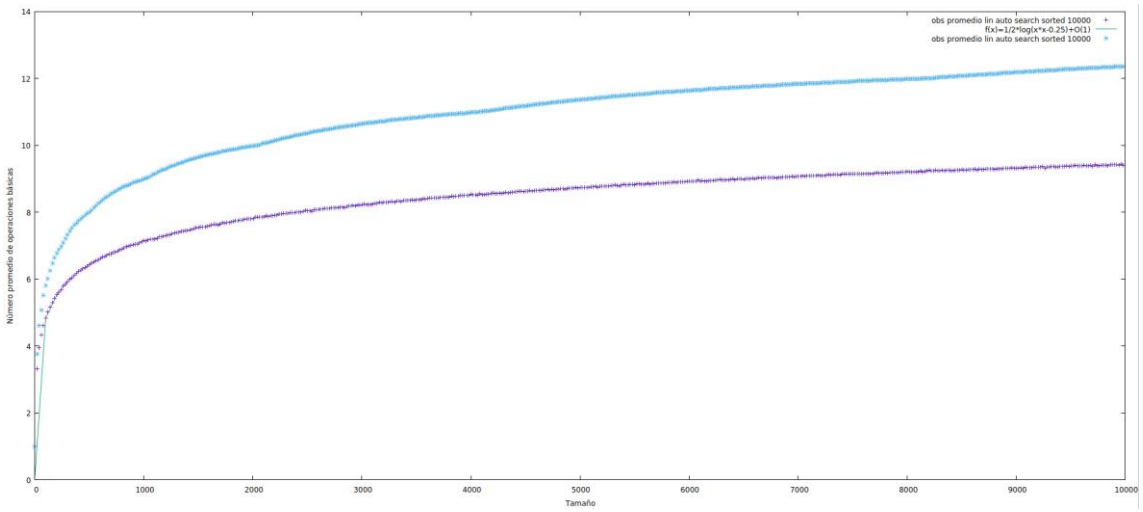
$$A_{LAS}(N, k) = \frac{1}{3} + \frac{1}{N} + \sum_{i=2}^{N-1} \frac{i}{i^2 - \frac{1}{4}} + \left(\frac{2}{2N - 1} - \frac{1}{N} \right) \cdot N$$

Realizando las determinadas cuentas para aproximar ese sumatorio por integrales, llegamos a que (log es el logaritmo en base e):

$$\boldsymbol{A_{LAS}(N, k) = \frac{1}{2} \log\left(N^2 - \frac{1}{4} \right) + O(1) = \frac{1}{lg(e^2)} \log\left(N^2 - \frac{1}{4} \right) + O(1)}$$

Veamos que se cumple para tablas ordenadas. Para ello, modificamos el exercise2 para que genere diccionarios de tipo SORTED y lin_auto_search con el generador potencial. Con un fit que ayude a ajustar con esa O(1), tenemos la siguiente gráfica





Tal y como se predijo en el apartado de las gráficas, lin_auto_search consigue un rendimiento mejor una vez ha alcanzado la estabilidad. Esa diferencia es de aproximadamente 1/lg(e).

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

Sea T una tabla ordenada y sea T[M] el elemento situado justo en el centro de la tabla o en la parte entera de la mitad entre los límites superior e inferior. Eso significa que todos los elementos que están a su izquierda son menores a él y todos los que están a su derecha son mayores. Si preguntamos por un elemento y resulta coincidir con él, entonces evidentemente lo encuentra y lo devuelva.

Si nuestro elemento a buscar es menor que él, entonces buscará en la subtabla izquierda. Por contra, si es mayor, buscará en la subtabla de la derecha. Así pues, lo que hay que hacer es ir actualizando los límites superior e inferior. Sean F (el inferior) y L (el superior),

entonces la subtabla de la izquierda tiene como límite inferior F y como límite superior M-1, con M=(L+F)/2 (parte entera). Por tanto, habría que actualizar el límite superior y repetir el mismo argumento en esa subtabla.

En el otro caso, habría que actualizar el límite inferior a M+1, siendo el superior L. Entonces, habría que repetir el proceso, pero en la subtabla de la derecha.

Si no lo encuentra, llegará un punto en que lleguemos a una subtabla en que el límite inferior sea mayor que el superior. Eso se producirá cuando tengamos que buscar entre medias de dos elementos y la actualización de los límites superior e inferior nos lleve a un error.

7. Conclusiones finales.

En esta práctica, hemos comparado los diferentes algoritmos de búsqueda. Nos ha sorprendido ver lo eficiente que es tener un algoritmo logarítmico en comparación con un algoritmo lineal. Ha sido justo a raíz de realizar esta práctica cuando hemos podido ver de primera mano cómo de ineficiente (en comparativa claro) es un algoritmo lineal.

Por otro lado, cabe mencionar también lo de lin_auto_search, que tal y como hemos demostrado en la cuestión propia de ello, puede llegar a conseguir un rendimiento bastante sorprendente cuando consigue una tabla que se ajusta a la distribución de probabilidad que una generación potencial produce. Un rendimiento incluso mejor que el propio bin_search.