Análisis de Algoritmos 2022/2023

Práctica 3

Diego Rodríguez Ortiz y Alejandro García Hernando, Grupo 1202.

Código	Gráficas	Memoria	Total

1. Introducción.

A lo largo de esta práctica hemos trabajado con la estructura *DICT* y experimentado con sus distintas utilidades. Hemos distinguido las funciones según su propósito en dos ficheros: en los ficheros *search* se encuentran todas las rutinas encargadas de la inicialización del diccionario, de buscar en él con distintos métodos y de insertar los distintos elementos. Por otro lado, en los ficheros *times* se encuentran aquellas rutinas que miden los tiempos de las descritas anteriormente.

2. Objetivos

2.1 Apartado 1

En este primer apartado desarrollamos las funciones *bin_search, lin_search y lin_auto_search* que hacen uso de los algoritmos que les dan nombre para buscar en el diccionario. Además, para trabajar con la estructura DICT hay que construir las funciones pertinentes que lo inicialicen, liberen, inserten y busquen en él.

2.2 Apartado 2

Debemos crear todas las rutinas que se encargan de medir el tiempo empleado por las funciones anteriores para encontrar en un diccionario. Average_search_time se encarga de medir cuánto tarda en buscar en un diccionario un determinado número de veces cada elemento de la tabla; posteriormente se almacenarán estos datos en una tabla de tipo TIME_AA. También hay que crear generate_search_times, que llamará a la función anterior con cada uno de los tamaños indicados.

3. Herramientas y metodología

Cómo ambos tenemos ordenadores Windows, hemos decidido programar en Visual Code y ejecutar nuestros programas en Ubuntu. Además, para realizar las gráficas hemos empleado gnuplot y hemos comprobado que no haya fugas de memoria con Valgrind.

3.1 Apartado 1

Para desarrollar las funciones relacionadas con la creación, impresión, inserción y liberación de diccionarios nos hemos basado en la estructura de datos facilitada. Por otro lado, para crear las rutinas de búsqueda hemos creado una general que se encargue de llamar a la bin_search, lin_search o lin_auto_search en función del tipo de búsqueda deseada. Y para finalizar hemos construido las 3 funciones de búsqueda citadas anteriormente.

3.2 Apartado 2

Para escribir las rutinas que midieran el tiempo, hemos usado numerosas funciones de la práctica 1 relacionadas con la generación de permutaciones del tamaño indicado. Además, para desarrollar generate_search_time hemos necesitado construir average_search_time, que se encarga de almacenar la información relacionada con la búsqueda de claves con el método indicado en los parámetros de entrada. Posteriormente hemos plasmado los datos, facilitados al ejecutar los distintos archivos exercise, con gnuplot.

4. Código fuente

4.1 Apartado 1

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

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

  DICT=(PDICT)calloc(1,sizeof(DICT));
  if(DICT==NULL){
    return NULL;
  }
  DICT->size=size;
  DICT->n_data=0;
  DICT->order=order;
  DICT->table=(int*)calloc(size,sizeof(int));
  if(DICT->table==NULL){
    free(DICT);
    return NULL;
  }
  return DICT;
  /* your code */
}
void print_dict(PDICT pdict){
  int i;
  for (i=0;i<pdict->n_data;i++){
    printf(" %d ",pdict->table[i]);
  }
}
void free_dictionary(PDICT pdict)
{
  if(!pdict) return;

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

```c
int insert_dictionary(PDICT pdict, int key)
{
  int i;
  if(!pdict) return ERR;
  if(pdict->size==pdict->n_data) return ERR;/*Comprobamos que no este lleno*/

  if(pdict->n_data==0){ /*Si el diccionario está vacío*/
    pdict->table[0]=key;
    pdict->n_data++;
    return 0;
  }

  if(pdict->order==NOT_SORTED){
    pdict->table[pdict->n_data]=key;
    pdict->n_data++;

    return 0;
  }
  else{
    i=pdict->n_data-1;
    if(pdict->table[i]<key){/*Es mayor que el ultimo*/
      pdict->table[i+1]=key;
      pdict->n_data++;

      return 1;
    }
    while(i>=0 && pdict->table[i]>key){
      pdict->table[i+1]=pdict->table[i];
      i--;
    }
    if(i<0) i++;
    pdict->table[i]=key;
  }

  pdict->n_data++;

  return pdict->n_data-i-1;
}
```

```c
int massive_insertion_dictionary (PDICT pdict,int *keys, int n_keys)
{
  int i,count=0,nob;
  if(!pdict||!keys||n_keys<0){
    return ERR;
  }
  for(i=0;i<n_keys;i++){


    nob=insert_dictionary(pdict,keys[i]);
    if(nob<0){
      return ERR;
    }
    count+=nob;
  }
  return count;
}
```

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

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

  if(F==L){ /*Cuando solo queda una clave*/
    if(table[F]==key){
      *ppos=F;
      return 1;
    }
    *ppos=NOT_FOUND;
    return 1;
  }

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

```c
int lin_search(int *table,int F,int L,int key, int *ppos)
{
  int i;
  if(!table|| F>L || !ppos)  return ERR;

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


int lin_auto_search(int *table,int F,int L,int key, int *ppos)
{
  int i;
  if(!table||F>L||!ppos) return ERR;

  if(table[F]==key){
      *ppos=F;
      return 1;
  }

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

  *ppos=NOT_FOUND;
  return L-F+1;
}
```

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){
  int i, *table,*keys, pos,max_ob,min_ob,n_ob;
  long count=0;
  PDICT dict;
  clock_t t1, t2;
  if(!method||!generator||!ptime||N<0||(((int)order!=SORTED)&&((int)order!=NOT_SORTED))) return ERR;
  dict=init_dictionary(N,order);
  if(!dict) return ERR;

  if(order==SORTED){/*Tabla ordenada*/
    table=(int*)calloc(N,sizeof(int));
    if(!table) return ERR;
    uniform_key_generator(table,N,N);
    (void)massive_insertion_dictionary (dict,table,N);
    free(table);
  }
  else{ /*Tabla desordenada*/
    table=generate_perm(N);
    if(!table) return ERR;

    (void)massive_insertion_dictionary (dict, table, N);
    free(table);
  }

  keys=(int*)calloc(N*n_times,sizeof(int));
  if(!keys)  return ERR;
  generator(keys,n_times*N,N);

  t1 = clock();

  count=search_dictionary(dict,keys[0],&pos,method);
  max_ob=count;
  min_ob=count;
```

```c
  for(i=1;i<n_times*N;i++){

    n_ob=search_dictionary(dict,keys[i],&pos,method);
    if(n_ob<0){
      printf("Caso: %d %d %d ", i, keys[i], n_ob);
    }
    if(n_ob>max_ob){
      max_ob=n_ob;
    }
    if(n_ob<min_ob){
      min_ob=n_ob;
    }
    count+=n_ob;

  }
  t2 = clock();

  ptime->time = (double)(t2 - t1) / ((double)n_times*N);
  ptime->N=N;
  ptime->n_elems= n_times*N;
  ptime->max_ob=max_ob;
  ptime->min_ob=min_ob;
  ptime->average_ob=(double)count/(double)(n_times*N);
  free_dictionary(dict);
  free(keys);
  return OK;
}
```

```c
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){
  PTIME_AA ptime=NULL;
  int i,j,tam;
  if(!method||!generator||!file||num_min<0||num_max<num_min||n_times<1) return ERR;

  tam = (num_max - num_min) / incr + 1; /*Reserva dinamica de la tabla de datos*/
  ptime = (TIME_AA *)calloc(tam, sizeof(TIME_AA)); /*Reserva de memoria*/
  if(!ptime){
    return ERR;
  }

  for(i=num_min,j=0; i<=num_max && j<tam; i+=incr,j++){
    if(ERR==average_search_time(method, generator, (char)order, i, n_times, &ptime[j])){
      free(ptime);
      return ERR;
    }
  }
  if(ERR==save_time_table(file,ptime, tam)){
    free(ptime);
    return ERR;
  }
 free(ptime);  /*Posible error*/

  return OK;
}
```

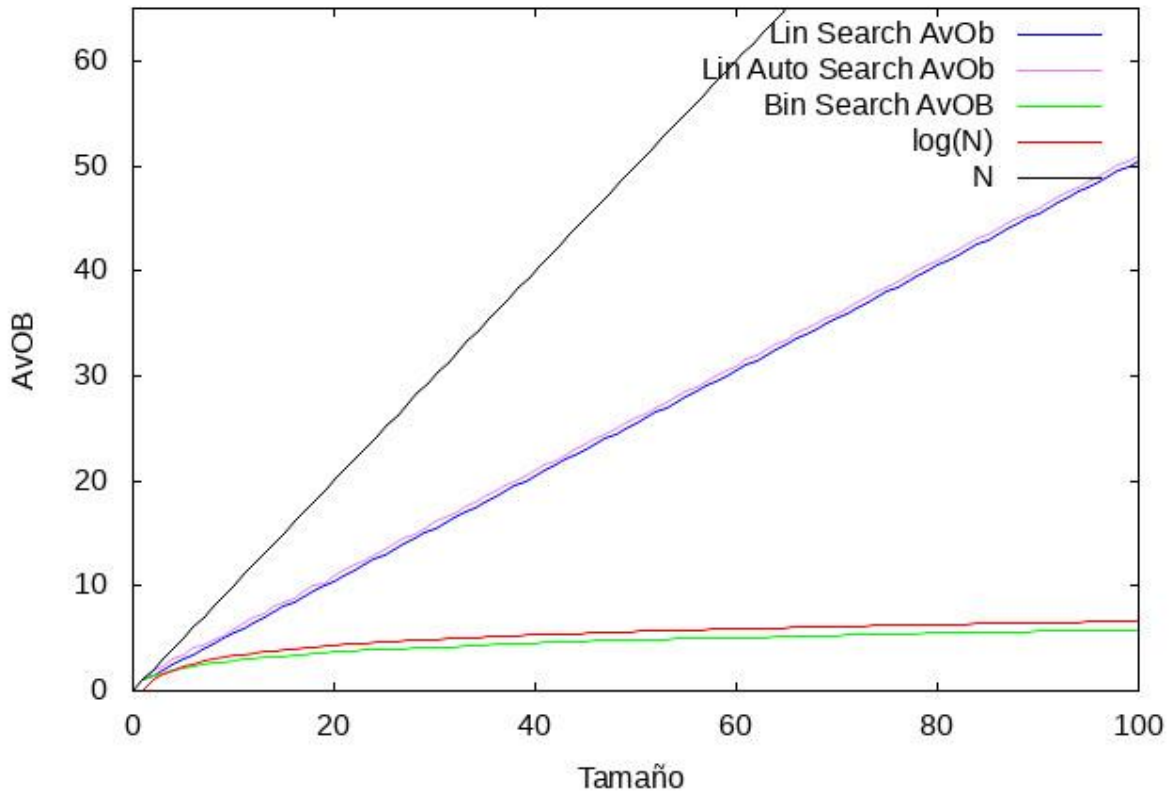
5. Resultados, Gráficas

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

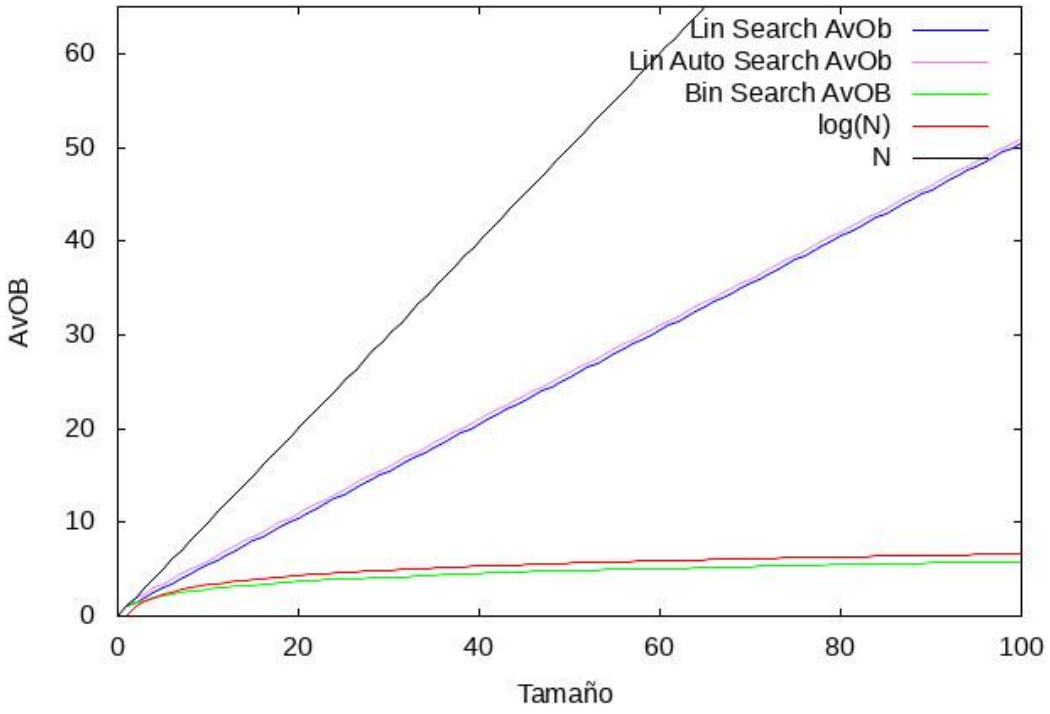
-Generación de claves uniforme:

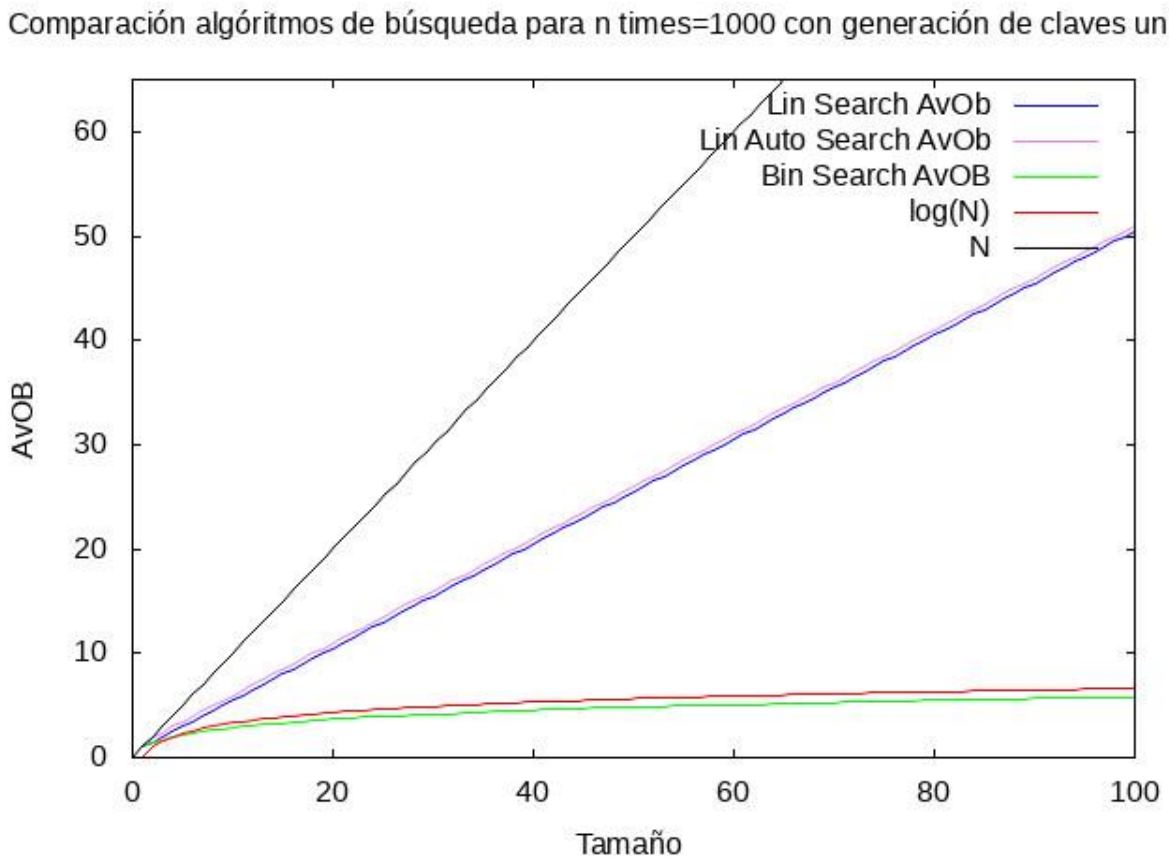
Cómo puede verse a continuación, cuando las claves se generan uniformemente se puede apreciar que el número medio de operaciones básicas realizadas de los 3 algoritmos es indiferente del número de veces que se busque cada clave. Además tanto Lin_Search como Lin_Auto_Search realizan algo menos de N OBs; mientras que Bin_Search realiza muchas menos, aproximadamente log(N).

Comparación algóritmos de búsqueda para n times=1 con generación de claves uniforr



Comparación algóritmos de búsqueda para n times=100 con generación de claves unifo





-Generador de claves potencial:

En este apartado generamos las claves con una distribución muy similar a la potencial, de tal forma que los valores menores son muchos más probables (para tamaño 100 el valor 1 tiene un 34.3%, el 2 un 22.6% y el 3 un 11.1%),

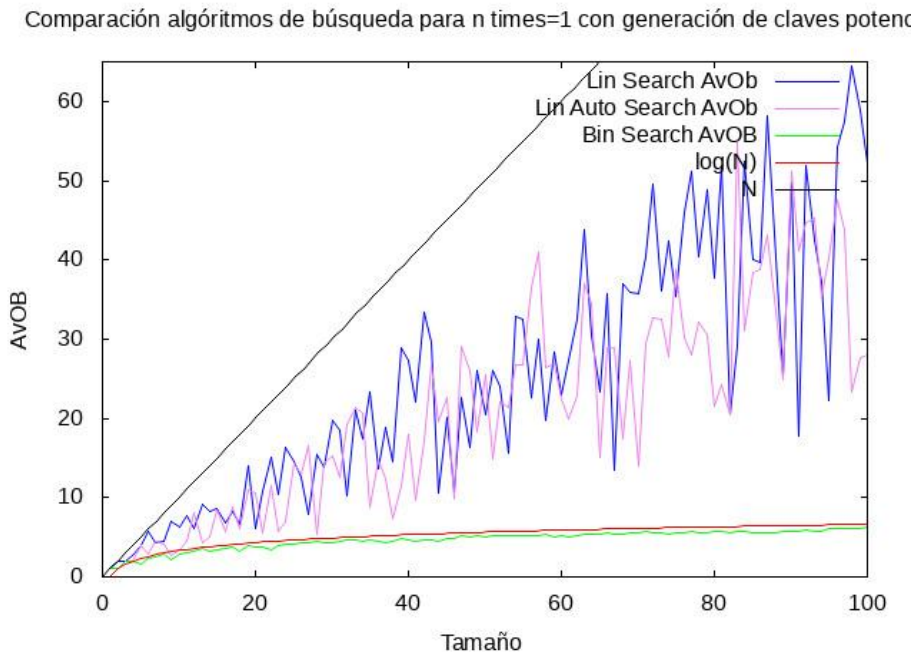
Por ello, cuando las claves se generan de forma potencial, el rendimiento de Bin_Search es similar a cuando se genera uniformemente y la influencia del número de veces que se busca cada clave es mínima. En cambio Lin_Auto_Search y Lin Search presentan una gran diferencia. En primer lugar, para Lin Auto Search cuánto mayor es el número de veces que se busca cada clave mejor es su rendimiento. Esto se debe a que se va cambiando el orden de los elementos de la lista, situando al principio aquellos que se busquen más veces. Dicho de otro modo, se necesitará realizar muy pocas operaciones básicas para encontrar aquellos elementos que tengan muchas probabilidades de ser buscados, en este caso los de menor valor, bajando drásticamente el número medio de OBs realizadas.

Cuanto más grande sea n_times, más ordenada se encontrará la lista, pues más claves se buscarán y las más probables irán avanzando poco a poco. También hay que tener en cuenta que cuanto mayor sea el tamaño del diccionario, más búsquedas se

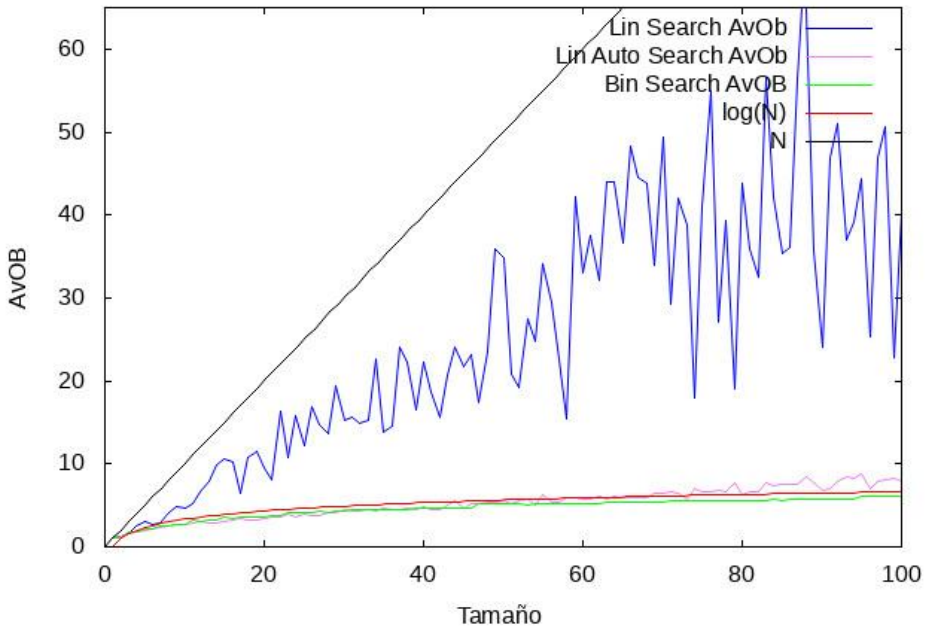
necesitarán para ordenarlo. Esta situación puede verse en la cuarta gráfica, donde a pesar de que n_times=10000, no han sido suficientes para ordenarla y alcanzar su pico de eficiencia debido al gran tamaño de la lista.

Por otro lado, Lin Search (y Lin Auto Search para n_times pequeños) presenta grandes fluctuaciones. Las variaciones para Lin Auto Search se mitigan para n_times grandes , ya que al generarse N*n_times claves de forma potencial es suficiente para ordenar la tabla, haciendo que la situación inicial del diccionario apenas influya en el promedio de operaciones básicas realizadas. En estos casos, cabe resaltar la estabilidad del crecimiento de las operaciones básicas medias ejecutadas por Lin_Auto_Search y Bin_Search mientras que el de Lin_Search continúa siendo extremadamente irregular.

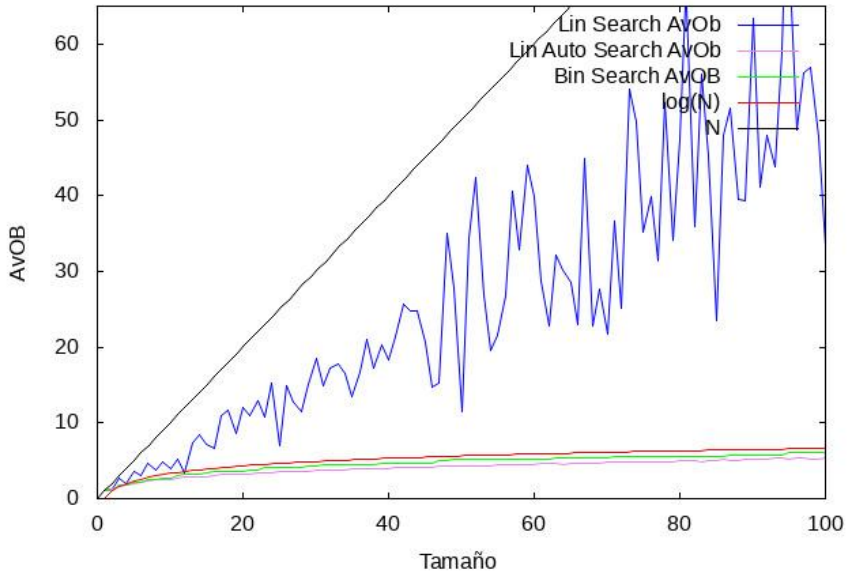
Bin_Search necesita entre 1 y log(N) OBs para encontrar un elemento. Esto hace que su coste medio apenas se vea influenciado por la situación de la lista. Falta comentar el caso más complejo e interesante de los 3, Lin_Search, que se ve influenciado en gran medida por la posición inicial de los números bajos dentro de la lista; por ello, dado que se genera una única permutación para cada tamaño y la posición de los números más probables varía, el promedio de OBs realizadas para cada una de ellas varía enormemente.

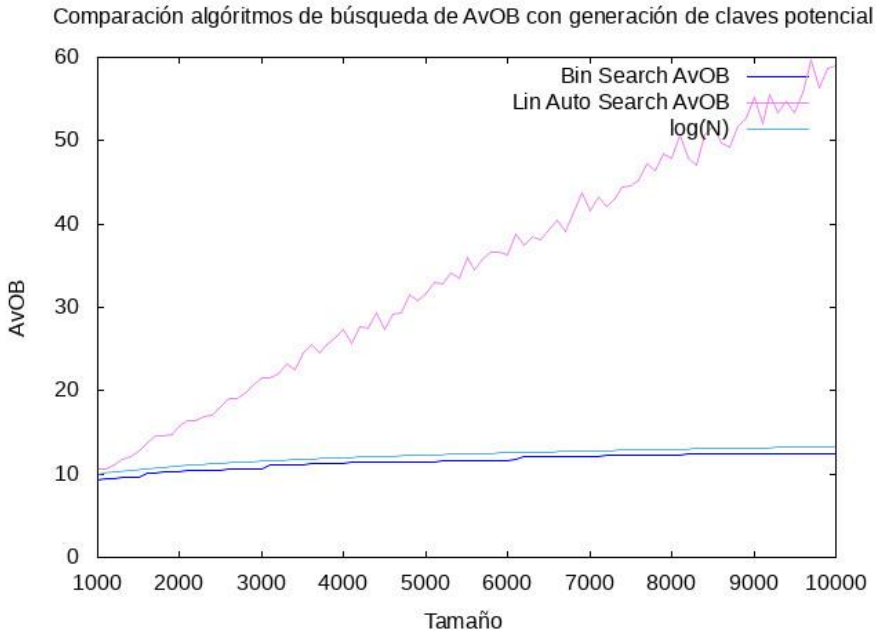


Comparación algóritmos de búsqueda para n times=100 con generación de claves poter



Comparación algóritmos de búsqueda para n times=1000 con generación de claves pote

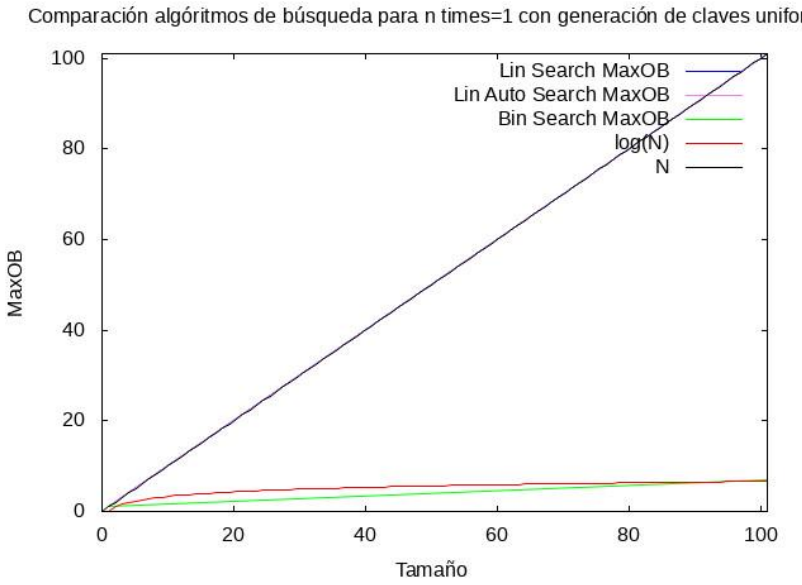


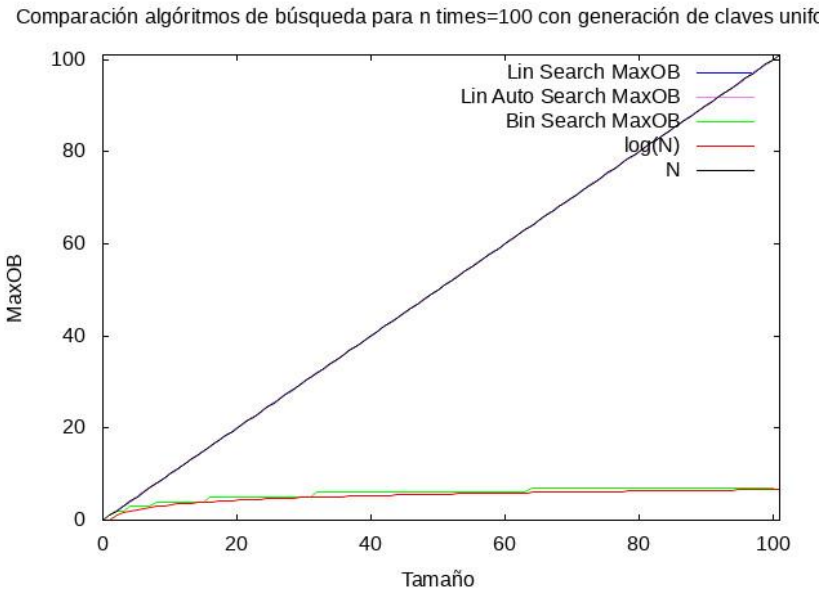


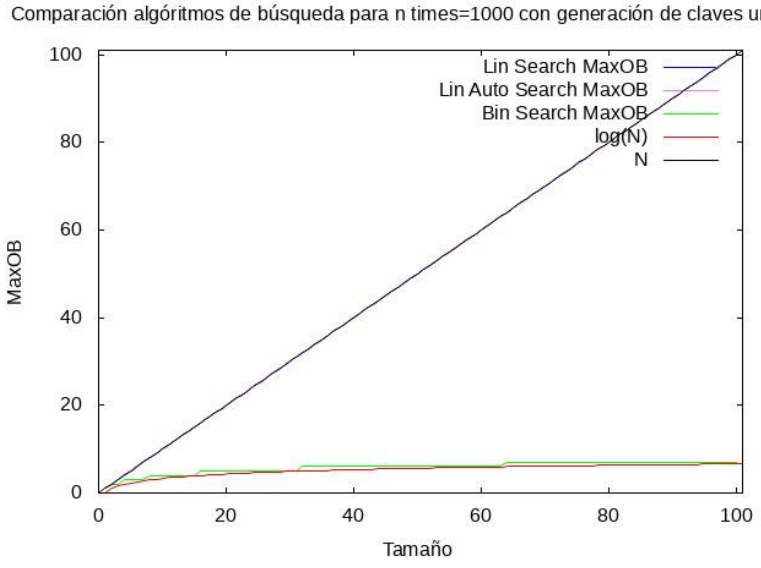
Gráfica comparando el número máximo de OBs entre la búsqueda binaria y la búsqueda lineal auto organizada (para los valores de n_times=1, 100 y 10000), comentarios a la gráfica.

-Generador de claves Uniforme:

Las gráficas nos muestran que cuando las claves son generadas uniformemente el número máximo de OBs no se ve afectado por el número de veces que se busque cada clave, esto se debe a que se buscarán todas las claves y como resultado el caso peor de los 3 algoritmos siempre aparecerá. En el caso de Lin_Search y Lin_Auto_Search se realizan como máximo N operaciones básicas, pues su caso peor es cuando se busca el elemento que se encuentra al final del diccionario, necesitando comparar la clave con todos los elementos anteriores para encontrarlo. Por otro lado, Bin_Search necesita como mucho ⌈log(N)⌉ comparaciones de claves para encontrar cualquier elemento, sin importar el número de veces que se busque cada clave.


Comparación algóritmos de búsqueda para n times=1 con generación de claves unifor
Lin Search MaxOB
Lin Auto Search MaxOB
Bin Search MaxOB
log(N)
N
MaxOB
Tamaño
0
20
40
60
80
100

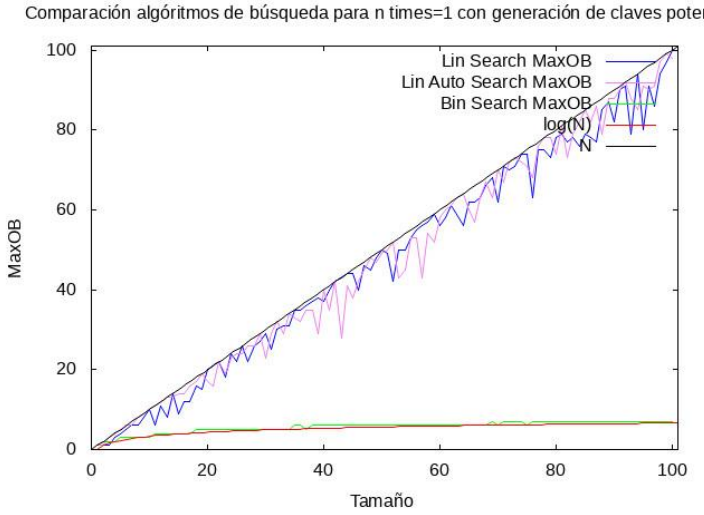

Comparación algóritmos de búsqueda para n times=100 con generación de claves unifc
Lin Search MaxOB
Lin Auto Search MaxOB
Bin Search MaxOB
log(N)
N
MaxOB
Tamaño
0
20
40
60
80
100

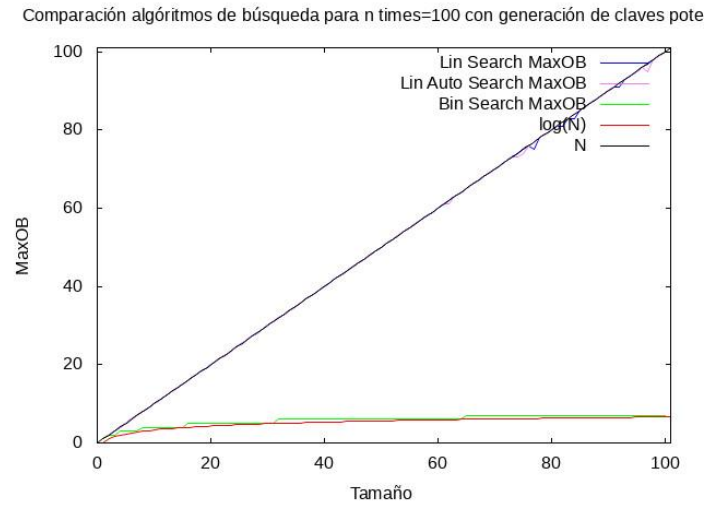

Comparación algóritmos de búsqueda para n times=1000 con generación de claves unif
Lin Search MaxOB
Lin Auto Search MaxOB
Bin Search MaxOB
log(N)
N
MaxOB
Tamaño
0
20
40
60
80
100

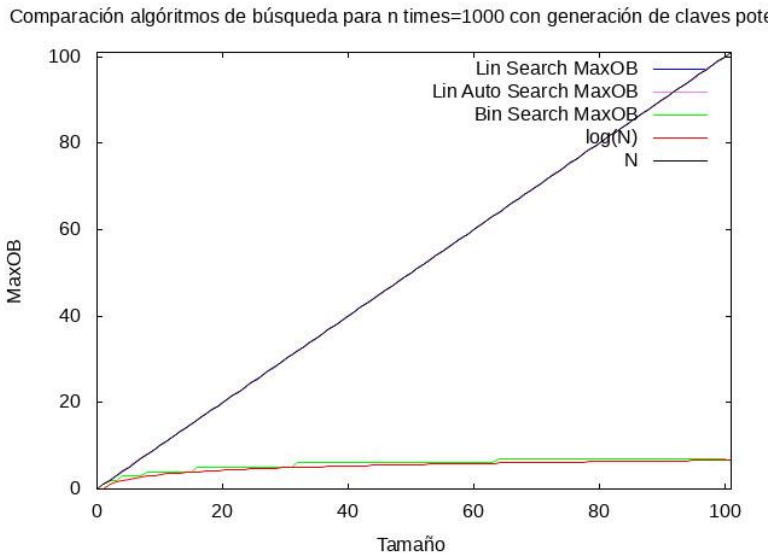
-Generador de claves Potencial:

Las operaciones máximas realizadas cuando las claves son generadas de forma potencial varía drásticamente en función de n_times. Para entender qué causa esta situación es importante saber que cuanto más cerca del finalde la lista esté una clave, más OBs tendrán que hacer Lin_Search y Lin_Auto_Search para encontrarla. Por ello deducimos que el peor caso es la clave en la posición N.

Sabiendo que se generan N*n_times claves, si n_times es un número bajo (como 1) se generan muy pocas claves, que sumado a la baja probabilidad de que aparezcan números altos, es muy probable que el número de OBs máximas realizadas para buscar las claves generadas, no sea cercano al peor pues se encuentra muy influenciado por el estado inicial de la tabla. Esto genera grandes fluctuaciones pues para cada tamaño de permutación se genera un conjunto distinto, y depende de la suerte que el peor caso de las claves generadas sea cercano al peor.



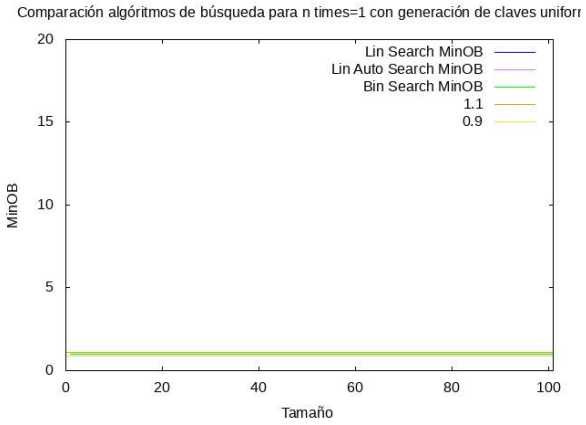


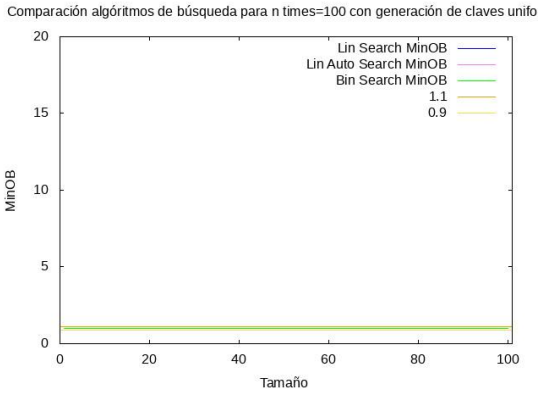


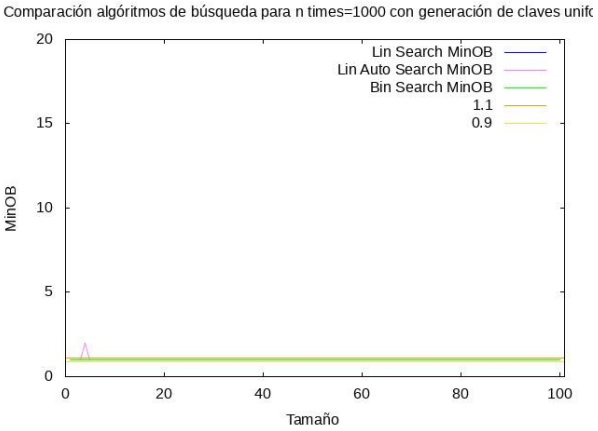
Gráfica comparando el número mínimo de OBs entre la búsqueda binaria y la búsqueda lineal auto organizada (para los valores de n_times=1, 100 y 10000), comentarios a la gráfica.

-Generador de claves Uniforme:

El mejor caso de Lin_Search y Lin_Auto_Search es cuando se busca el valor en la posición 1, y el de Bin_Search es buscar el elemento en la posición N/2, para todos ello conlleva únicamente una comparación de claves. Cuando la generación de claves es uniforme y se generan al menos N claves (como sucede en los casos estudiados en esta práctica), el conjunto de claves generadas siempre contendrán los mejores casos de los tres algoritmos. Por ello, el mínimo de OBs que realizan los algoritmos Bin search y Lin search será 1, sin importar cuántas veces se busque cada clave. Sin embargo, como Lin auto search altera las posiciones de la tabla existe la posibilidad de que ninguna búsqueda haga una sola OB. Toma el ejemplo de la tabla 1 2 3, y se generan las claves (2,1,3) en ese orden. Tras buscar el 2 (2 comparaciones de claves) la tabla está en el siguiente estado 2 1 3. Posteriormente al buscar el 1 (otras 2 OBs) el diccionario vuelve a alterarse quedando 1 2 3. Finalmente, buscar el 3 realiza 3 comparaciones. Existen distintas combinaciones de claves generadas que producen este curioso y raro fenómeno, que tuvimos la fortuna de capturar en la tercera gráfica mostrada.



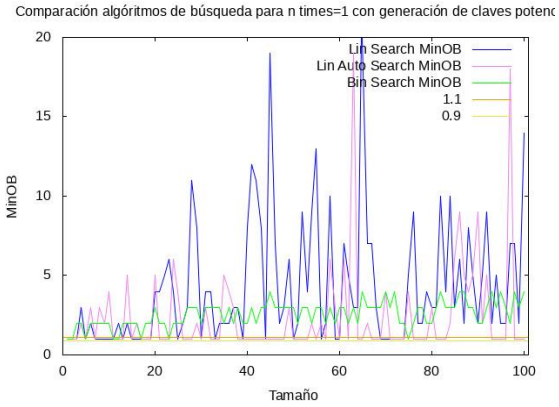


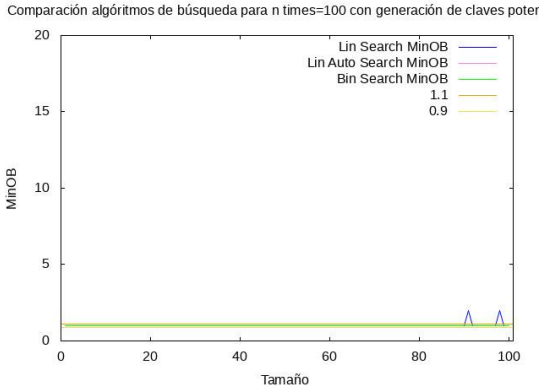


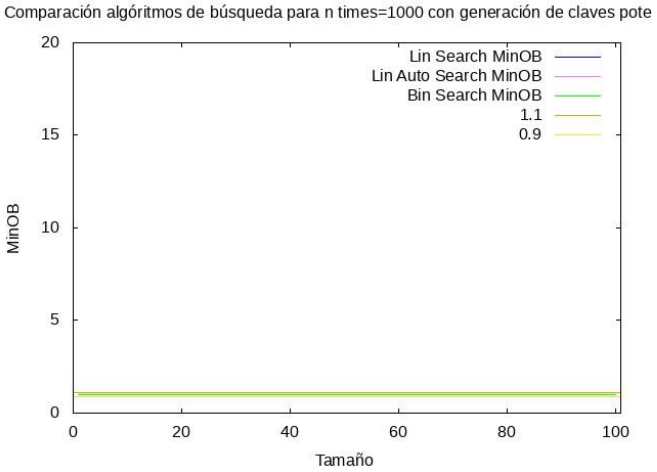
-Generador de claves Potencial:

Recordamos que el mejor caso de Lin_Search y Lin_Auto_Search es cuando se busca el valor en la posición 1, y el de Bin_Search es buscar el elemento en la posición N/2, para todos ellos se necesita únicamente 1 comparación de claves.

Cuando la generación de claves es potencial, lo más probable es que se generen números bajos, de hecho, si se generan N claves sabemos que más del 75% de ellas serán menores a 10. Con todo esto, deducimos que es muy probable que si n_times es bajo, el número mínimo de operaciones básicas ejecutadas dependerá mucho de la ordenación de los elementos y de las claves generadas en cada situación; siendo algo azarosa e impredecible. Por el contrario, si n_times es grande, la probabilidad de que se genere un caso *muy bueno* es mucho más alta, y los resultados serán mucho más predecibles y cercanos a la realidad. Todo esto puede ser corroborado con las gráficas inferiores obtenidas experimentalmente.



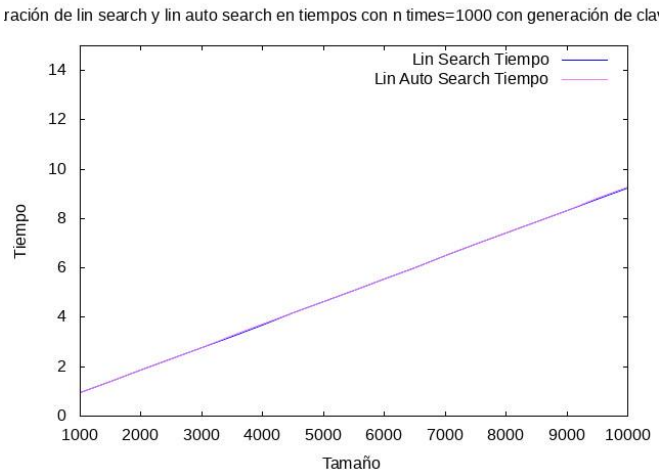


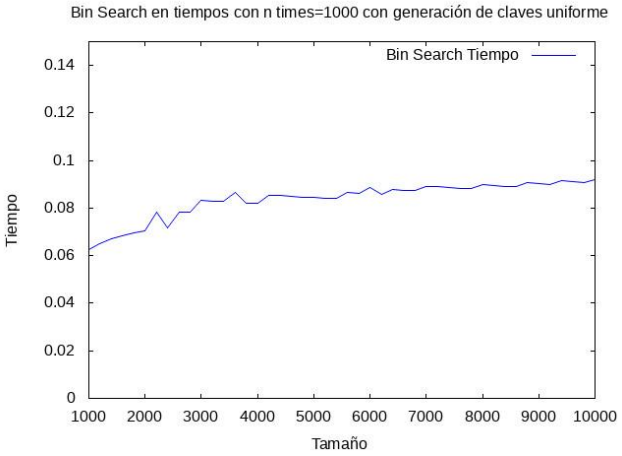


Gráfica comparando el tiempo promedio de reloj entre la búsqueda lineal, la búsqueda binaria y la búsqueda lineal auto organizada (para los valores de n_times=1000) ,comentarios a la gráfica.

-Generador de claves Uniforme:

Como cabría esperar tras el análisis de las operaciones básicas, si el generador de claves es uniforme, tanto lin search como lin auto search tiene un crecimiento lineal rondando valores entre las 1,5 y 15 unidades de tiempo, mientras que bin search es mucho más rápida apenas llegando a la 0,1 unidad de tiempo con diccionarios de 10000 elementos.

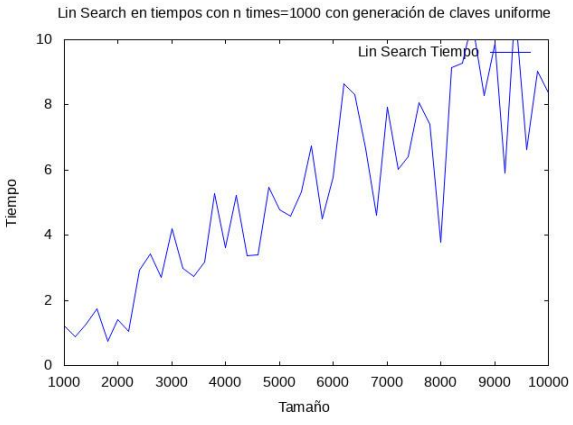


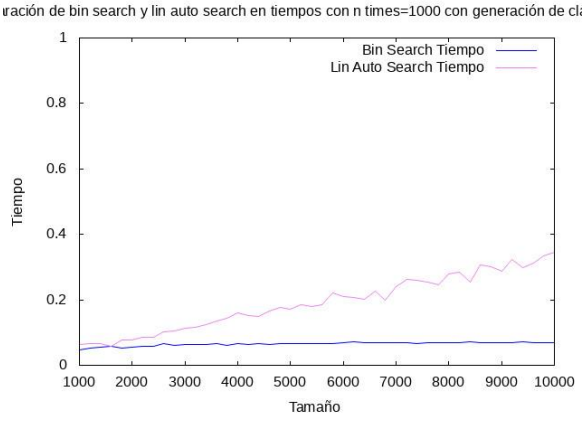


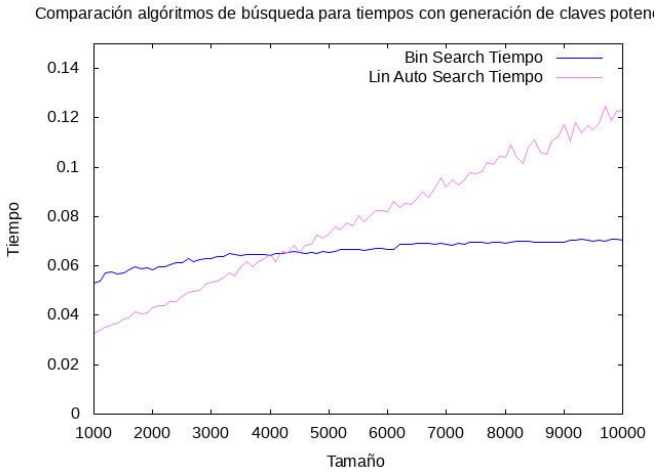
-Generador de claves Potencial:

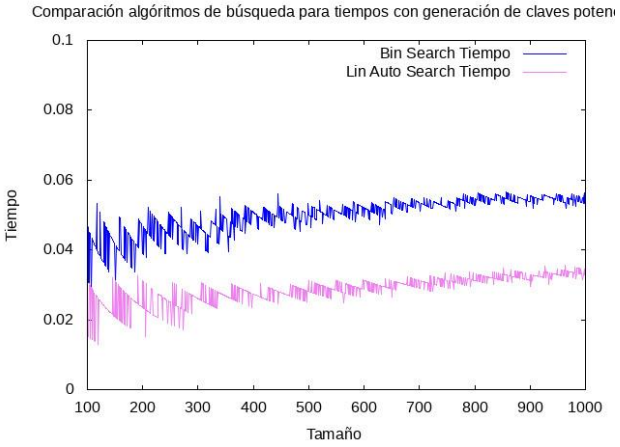
De nuevo las gráficas de tiempo son similares a la de OBs, viendo en el algoritmo lin search las ya mencionadas fluctuaciones para esta distribución. Por otro lado vemos como el coste de lin auto search ha pasado de ser similar a lin search, a ser cercano al de bin search.

Vemos en la cuarta imagen que cuando aumentamos aún más n times hasta 10000 para tamaños pequeños (100-1000) vemos como el coste de lin auto search está por debajo del de bin search. Esto parece sorprendente pero hay que tener en cuenta que para que este algoritmo alcance toda su eficacia se necesitan un gran número de n times. De hecho, como hemos comentado con anterioridad, el número de n times para que lin auto search ordene el diccionario (estado en el que el algoritmo es más eficiente, a veces incluso más que bin search) depende de forma directa de N, como puede verse en la segunda gráfica, donde usamos las mismas n times pero diccionarios mayores.







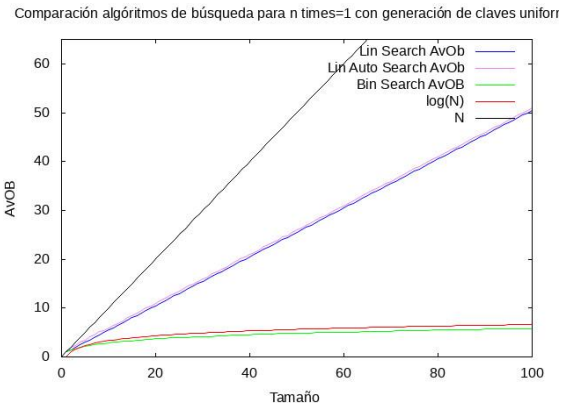


N times=10000

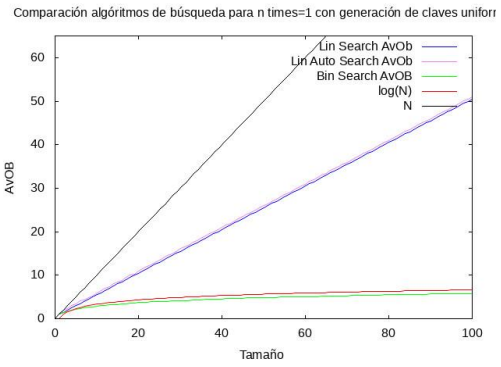
Para finalizar el análisis de la gráficas nos gustaría añadir que si utilizamos los algoritmos anteriores para buscar en varias permutaciones nos encontramos ante la siguiente situación:

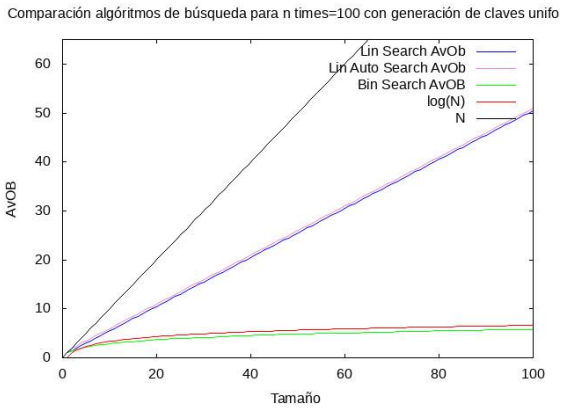
-Generador de claves Uniforme:

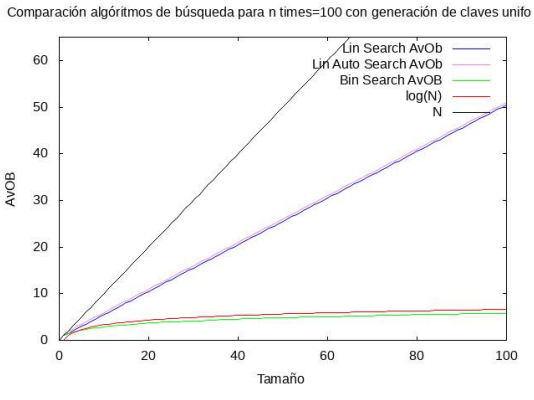
Sobre 1000 permutación:

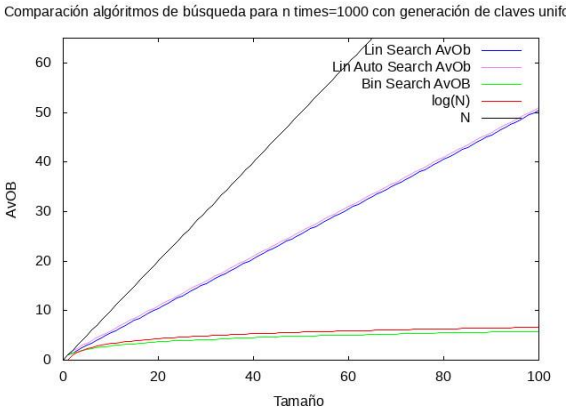


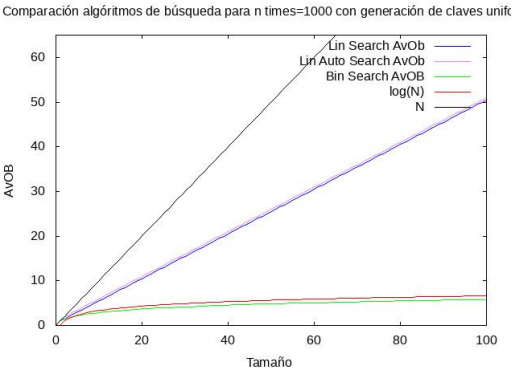
Sobre 1 permutación:









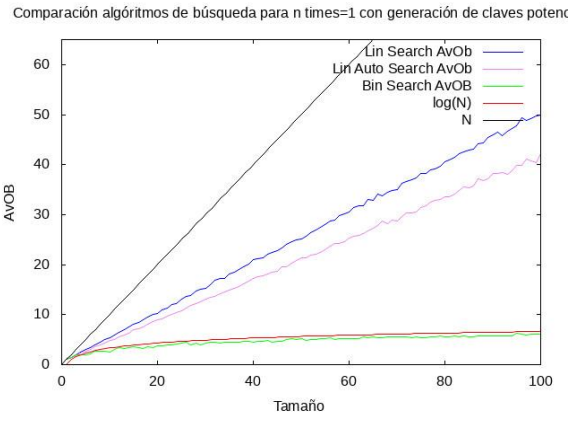


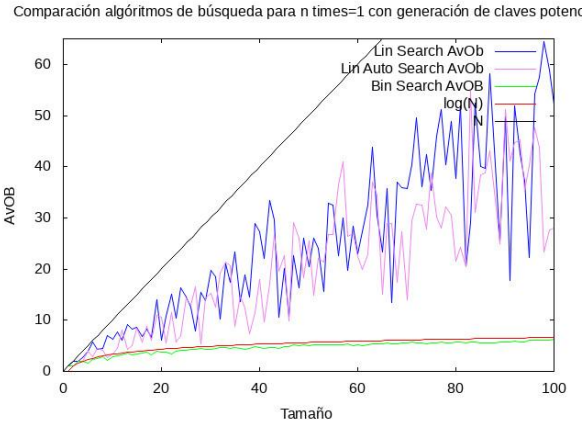
Como podemos ver en la gráficas adjuntas con anterioridad, si la generación de claves es uniforme las operaciones básicas para buscar con los 3 algoritmos no varían si trabajamos con una permutación o con varias.

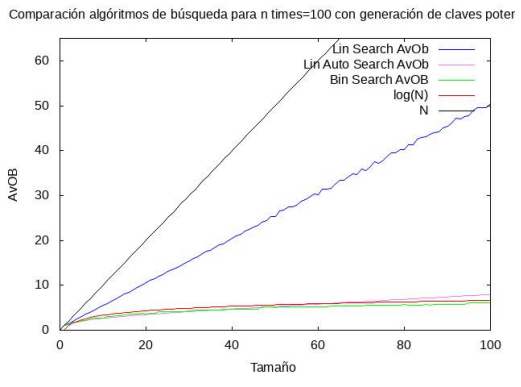
-Generador de claves Potencial

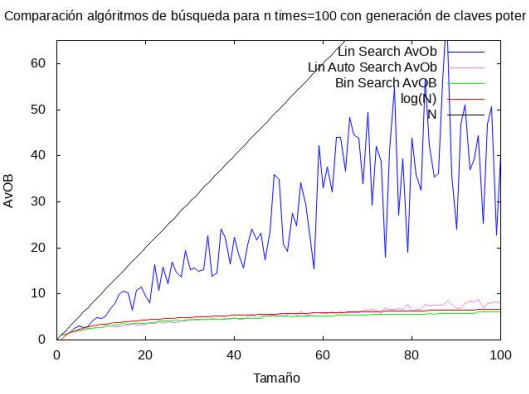
En la generación de claves potencial es donde se aprecia la importancia de medir las operaciones básicas medias de los distintos algoritmos sobre diversas permutaciones aleatorias. En las tres gráficas se ve cómo se mitigan las fluctuaciones de Lin Search mencionadas en las primeras gráficas, ya que la influencia de la posición de las claves más probables se disminuye al ejecutar el algoritmo sobre 1000 permutaciones distintas y hacer la media.

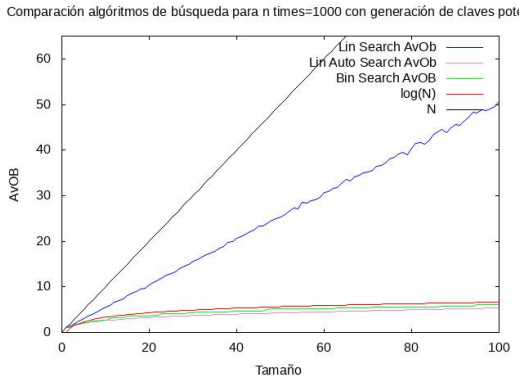
<u>Sobre 1000 permutación:</u> **<u>Sobre 1 permutación:</u>**

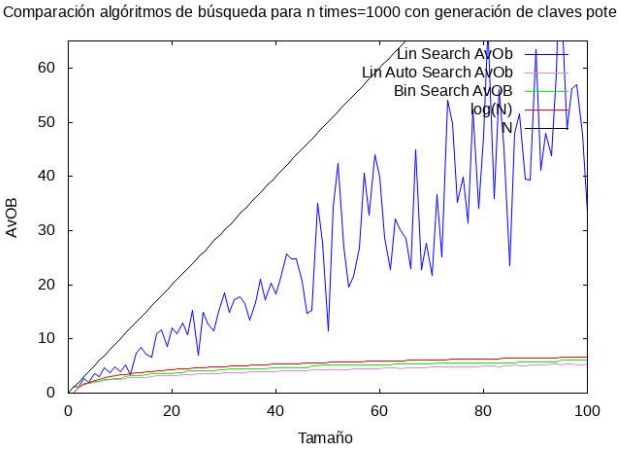












6. Respuesta a las preguntas teóricas.

6.1 ¿Cuál es la operación básica de bin_search, lin_search y lin_auto_search?

Al igual que en los algoritmos de ordenación estudiados en los análisis anteriores, la operación básica de bin_search, lin_search y lin_auto_search es la comparación de claves.

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.

Como hemos visto en teoría, los casos mejor y peor para bin_search y lin_search no son difíciles de calcular. Por un lado, para ambos algoritmos el caso mejor es Θ(1). De hecho realiza una única operación básica, ya que si la primera comparación se realiza con la clave que buscamos, el algoritmo parará pues ya la habrá encontrado. Por otro lado, el caso peor es muy distinto. Mientras que para bin search tiene un coste logarítmico log(N)+O(1) necesitando realizar ⌈log(N)⌉ operaciones básicas, lin_search tiene un coste lineal (Θ(N)) realizando exactamente N comparaciones de clave (esta situación se dará cuando la clave que se busca se encuentre al final del diccionario)

Debido a la simplicidad de los cálculos se puede extraer exactamente el número de OB que realizarán los algoritmos:

Bbs(N)=Bls(N)=1 Wbs(N)=⌈log(N)⌉ Wls(N)=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?

La idea de lin_auto_search es que cada vez que se encuentra una clave, ésta se coloca una posición más adelante en la tabla. Provocando que para futuras búsquedas, la misma clave se encuentre en menos tiempo. Tras realizar muchas búsqueda de la misma clave esta va avanzando hasta que se encuentra en las primeras posiciones del diccionario y encontrarla tan solo requiere unas pocas OBs.

Es fundamental que la distribución de las claves no sea uniforme, ya que si lo fuera, todas las claves avanzarían hacia delante al mismo ritmo, haciendo que ninguna clave obtenga una ventaja para la búsqueda y el coste del algoritmo de búsqueda sería lineal. Sin embargo, si la distribución no es uniforme significa que hay claves con más posibilidades de ser buscadas, haciendo que dichas claves avancen hacia el frente de la tabla.

En conclusión, lin_auto_search consigue que las claves más probables sean las mejores para buscar (porque se necesitan menos OBs para encontrarlas).

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.

Tras varias ejecuciones, la tabla seguirá una distribución similar a la de la probabilidad de las claves es decir: la clave 1 que tiene la mayor probabilidad de ser buscada será la primera, luego la 2, después la 3 etc. Es decir, la tabla estará ordenada en función de la probabilidad de generación de cada clave. En nuestro caso como hemos observado de forma empírica, lin_auto_search llega a rebajar el coste a logarítmico cuando el número de búsquedas pasa un cierto umbral, como vemos en la gráfica con n_times=1000. Por tanto podemos concluir, que el coste medio empírico de lin auto search es O(log(N)).

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

Bin_search es muy eficaz para listas ordenadas. Esto se debe a que utiliza el elemento central de la lista como pivote (sabiendo que todos los elementos que se encuentren a su izquierda serán menores que dicho número, y que aquellos que se encuentren a la derecha serán mayores), y compara el elemento que buscamos con el pivote. Si la clave buscada coincide con el valor del pivote, basta devolver la posición del mismo, en caso contrario seguiremos el mismo procedimiento con la subtabla izquierda si el elemento es menor que el pivote, y en la su tabla derecha si es mayor. Es correcto descartar los elementos de la subtabla del lado contrario, porque como se trabaja sobre una tabla ordenada, en caso de existir sólo

puede encontrarse a la derecha si es mayor que el pivote empleado, o en la izquierda si no lo es.

7. Conclusiones

Tras haber comparado los tres algoritmos de búsqueda, podemos concluir que hay un claro perdedor: lin search. Este es igual o peor que lin_auto_search para todos los casos, pero no tiene la ventaja de que para una distribución de claves no uniforme puede lograr costes medios muy buenos, incluso por debajo de logarítmicos. Sin embargo, aunque la distribución no sea uniforme, lin_auto_search sigue teniendo un caso peor lineal. Por último, bin search es sin duda el mejor algoritmo de búsqueda si se conoce que el diccionario está ordenado, con un caso medio y peor logarítmico. En definitiva, para diccionarios ordenados destaca bin_search y para diccionarios sin ordenar lin_auto_search, pues ambos son mejores o iguales que lin_search.