Análisis de Algoritmos 2015/2016

Práctica 3

Javier Gómez y Carlos Li, Grupo 1201.

Código	Gráficas	Memoria	Total

1. Introducción.

En esta práctica vamos a implementar los algoritmos de búsqueda lineal, búsqueda lineal auto-organizada y búsqueda binaria para poder probarlos y medir las operaciones básicas que requieren para encontrar un elemento en una tabla así como lo tiempos de ejecución de cada uno de dichos algoritmos. De esta forma, aprenderemos sobre su funcionamiento y sobre qué ventajas y desventajas presentan, de manera que también aprenderemos para qué situaciones los algoritmos son más efectivos.

2. Objetivos

2.1 Apartado 1

En este apartado hemos implementado un tipo abstracto de datos diccionario como una tabla (ordenada o desordenada).

También hemos implementado la estructura TIEMPO, (reciclándola de la práctica anterior) que emplearemos para medir los tiempos de ejecución, el número de operaciones básicas medio, máximo y mínimo.

Finalmente, hemos implementado los algoritmos de búsqueda binaria (Bbin), búsqueda lineal (Blin) y búsqueda lineal auto-organizada (Blin_auto).

2.2 Apartado 2

Hemos creado un nuevo módulo tiempos.c y tiempos.h donde hemos trasladado la estructura TIEMPO así como las funciones y los prototipos de funciones ya hechos que tuvieran que ver con dicha estructura. Además, hemos implementado funciones nuevas como genera_tiempos_búsqueda y tiempo_medio_busqueda que sirven para recoger los datos de tiempos de ejecución, número mínimo y máximo de operaciones básicas, etc. en una estructura de tipo TIEMPO, pero en este caso los métodos a examinar eran algoritmos de búsqueda.

3. Herramientas y metodología

Hemos usado Linux, Netbeans, Valgrind, Gnuplot y Gedit.

3.1 Apartado 1

En este apartado hemos implementado en el módulo de busqueda.c y búsqueda.h los algoritmos de búsqueda lineal, lineal auto-organizada y binaria, que serán utilizados más adelante. También aquí hemos implementado el TAD diccionario, con todas las funciones necesarias.

Finalmente hemos utilizado la función main del fichero ejercicio1.c para probar cada uno de los algoritmos de búsqueda en tablas ordenadas y desordenadas (según correspondiera), asegurando que el número de operaciones básicas fuera el correcto y que la clave que deseáramos buscar se encontraba (en caso de que existiera) en la

posición correcta. En otras palabras, hemos comprobado que los algoritmos implementados funcionaran tal y como se esperaba.

3.2 Apartado 2

En este apartado, hemos creado el módulo tiempos.c y tiempos.h que contendría las funciones necesarias para examinar los rendimientos de los algoritmos. Para que la estructura de nuestro programa fuera consistente y no nos diera errores al incluir las librerías, los tipos abstractos de datos y macros definidas fueron también trasladadas al fichero ordenar.h, que es el fichero más "externo" de todos.

La implementación de las nuevas funciones de tiempos.c es similar a las funciones de las prácticas anteriores (que comprobaban los rendimientos de los algoritmos de ordenación) con la salvedad de que en este caso nuestros algoritmos debían buscar cada una de las claves de una tabla en la tabla del diccionario. Estas tablas de claves podían tener una distribución uniforme de las claves (es decir, si tiene tamaño 11 y hay 10 claves, cada clave aparece al menos una vez) o una distribución potencial (es decir, los valores más bajos son más comunes en la tabla, teniendo la clave 1 un 50% de probabilidades de aparecer, la clave 2 un 17%, el tres un 9% etc.). Por este motivo, era necesario incluir el método de generador de claves como un argumento en la función. Además, era necesario que los máximos y los mínimos se guardaran en un array, pues en caso de que la tabla no fuera ordenada, se debía realizar la búsqueda para permutaciones aleatorias distintas para poder obtener más precisión en la medida.

Finalmente, empleando la función main del fichero ejercicio2.c, hemos recogido los datos del rendimiento de los algoritmos en ficheros de texto y hemos representado dichos datos en gráficas mediante gnuplot para una interpretación más clara y eficaz de los rendimientos de estos.

4. Código fuente

4.1 Apartado 1

Funciones del TAD diccionario:

```
PDICC ini_diccionario(int tamanio, char orden) {
   PDICC dick;
   dick = (PDICC) malloc(sizeof (DICC));
   if (dick == NULL)
      return NULL;
   dick->tamanio = tamanio;
   dick->orden = orden;
   dick->n_datos = 0;
   dick->tabla = (int*) malloc(sizeof (int)*tamanio);

   return dick;
}
```

```
void libera_diccionario(PDICC pdicc) {
   if (pdicc->tabla != NULL)
      free(pdicc->tabla);
   if (pdicc != NULL)
      free(pdicc);
}


int inserta_diccionario(PDICC pdicc, int clave) {
   int A, j, OB = 0;

   if (pdicc == NULL)
      return -1;
   if (pdicc->n_datos >= pdicc->tamanio)
      return -1;

   if (pdicc->orden == NO_ORDENADO) {
      pdicc->tabla[pdicc->n_datos] = clave;
      pdicc->n_datos++;
      OB++;
   } else if (pdicc->orden == ORDENADO) {
      pdicc->tabla[pdicc->n_datos] = clave;
      A = pdicc->tabla[pdicc->n_datos];
      for (j = pdicc->n_datos - 1; j >= 0 && pdicc->tabla[j] > A; j--) {
         OB++;
         swap(pdicc->tabla + (j + 1), pdicc->tabla + j);
      }
      pdicc->n_datos++;
   }
   return OB;
}


int insercion_masiva_diccionario(PDICC pdicc, int *claves, int n_claves) {
   int i, k = 0, OB = 0;
   if (pdicc == NULL || claves == NULL)
      return -1;
   for (i = 0; i < n_claves; i++) {
      k = inserta_diccionario(pdicc, claves[i]);
      if (k == -1)return -1;
      OB += k;
   }
   return OB;
}


int busca_diccionario(PDICC pdicc, int clave, int *ppos, pfunc_busqueda metodo) {
   if(!pdicc)return -1;;
   return metodo(pdicc->tabla, 0, pdicc->n_datos-1, clave, ppos);
```

```
}

void imprime_diccionario(PDICC pdicc) {
   int i;
   if(!pdicc)
      return;
   for(i=0;i<pdicc->n_datos;i++)
      printf("%d ",pdicc->tabla[i]);
   printf("\n");
}
```

Funciones de búsqueda:

```
int bbin(int *tabla, int P, int U, int clave, int *ppos) {
   int OB = 0;
   if (!tabla || !ppos)
      return -1;
   while (P <= U) {
      OB++;
      *ppos = (P + U) / 2;
      if (tabla[*ppos] == clave)
         return OB;
      else if (clave < tabla[*ppos])
         U = (*ppos) - 1;
      else
         P = (*ppos) + 1;
   }
   return NO_ENCONTRADO;
}


int blin(int *tabla, int P, int U, int clave, int *ppos) {
   int OB = 0, i;
   if (!tabla || !ppos)
      return -1;
   for (i = P; i <= U; i++){
                  OB++;
                  if (clave == tabla[i]) {
      *ppos = i;
      return OB;
      }
         }

   return NO_ENCONTRADO;
}


int blin_auto(int *tabla, int P, int U, int clave, int *ppos) {
   int OB = 0, i;
```

```
   if (!tabla || !ppos)
      return -1;
   for (i = P; i <= U; i++){
                  OB++;
      if (clave == tabla[i]) {
         *ppos = i;
                        if(i!=0)
                                 {
                  swap(tabla+i,tabla+(i-1));
                                 (*ppos)--;
                                 }
         return OB;
      }
         }
   return NO_ENCONTRADO;
}
```

4.2 Apartado 2

```
short genera_tiempos_busqueda(pfunc_busqueda metodo, pfunc_generador_claves
generador,

                        int orden, char* fichero,

                        int num_min, int num_max,

                        int incr, double fclaves, int n_ciclos){

         int i, k, N;

   PTIEMPO tiempo;



   N = (int) ceil(1 + ((double) num_max - (double) num_min) / (double) incr);

   tiempo = (PTIEMPO) malloc(sizeof (TIEMPO)*(N));

   if (!tiempo)return -1;



   for (k = 0, i = num_max; k < N && i >= num_min; k++, i -= incr) {

      tiempo_medio_busqueda(metodo, generador, orden, i, fclaves, n_ciclos,
&(tiempo[k]));

   }
```

```
    guarda_tabla_tiempos(fichero, tiempo, N);
    free(tiempo);
    return OK;
}


short tiempo_medio_busqueda(pfunc_busqueda metodo, pfunc_generador_claves
generador,
                            int orden,
                            int tamanio,
                            double fclaves,
                            int n_ciclos,
                            PTIEMPO ptiempo){
        int flag, *max, *min, i, k;
        int *claves, ppos;
        int **L;
    PDICC dic;
    double sum, OB = 0;
    double megamax = 0, megamin = 0;
    double t_medio = 0;
    struct timespec time, aux;


        min = (int*)malloc(sizeof(int)*n_ciclos);
        max = (int*)malloc(sizeof(int)*n_ciclos);
        claves = (int*)malloc(sizeof(int) * fclaves*tamanio);
        if(claves == NULL)
                return -1;
```

```
	generador(claves,fclaves*tamanio,tamanio);
	L = genera_permutaciones(n_ciclos,tamanio);
	if(L == NULL)
		return -1;

	for(i = 0; i < n_ciclos; i++)
	{
		dic = ini_diccionario(tamanio,orden);
		if(dic == NULL)
			return -1;
		flag = insercion_masiva_diccionario(dic,L[i],tamanio);
		if(flag == -1)
			return -1;
		max[i] = 0;
		min[i] = tamanio;
		clock_gettime(CLOCK_REALTIME, &aux);
		for(k = 0; k < fclaves*tamanio; k++)
			{
			sum=busca_diccionario(dic, claves[k], &ppos, metodo);
			/*sum = metodo(dic->tabla, 0, (dic->tamanio)-1,
claves[k],&ppos);*/
			OB += sum;
			if (min[i] > (int)sum)min[i] = (int)sum;
	if (max[i] < (int)sum)max[i] = (int)sum;
			}
```

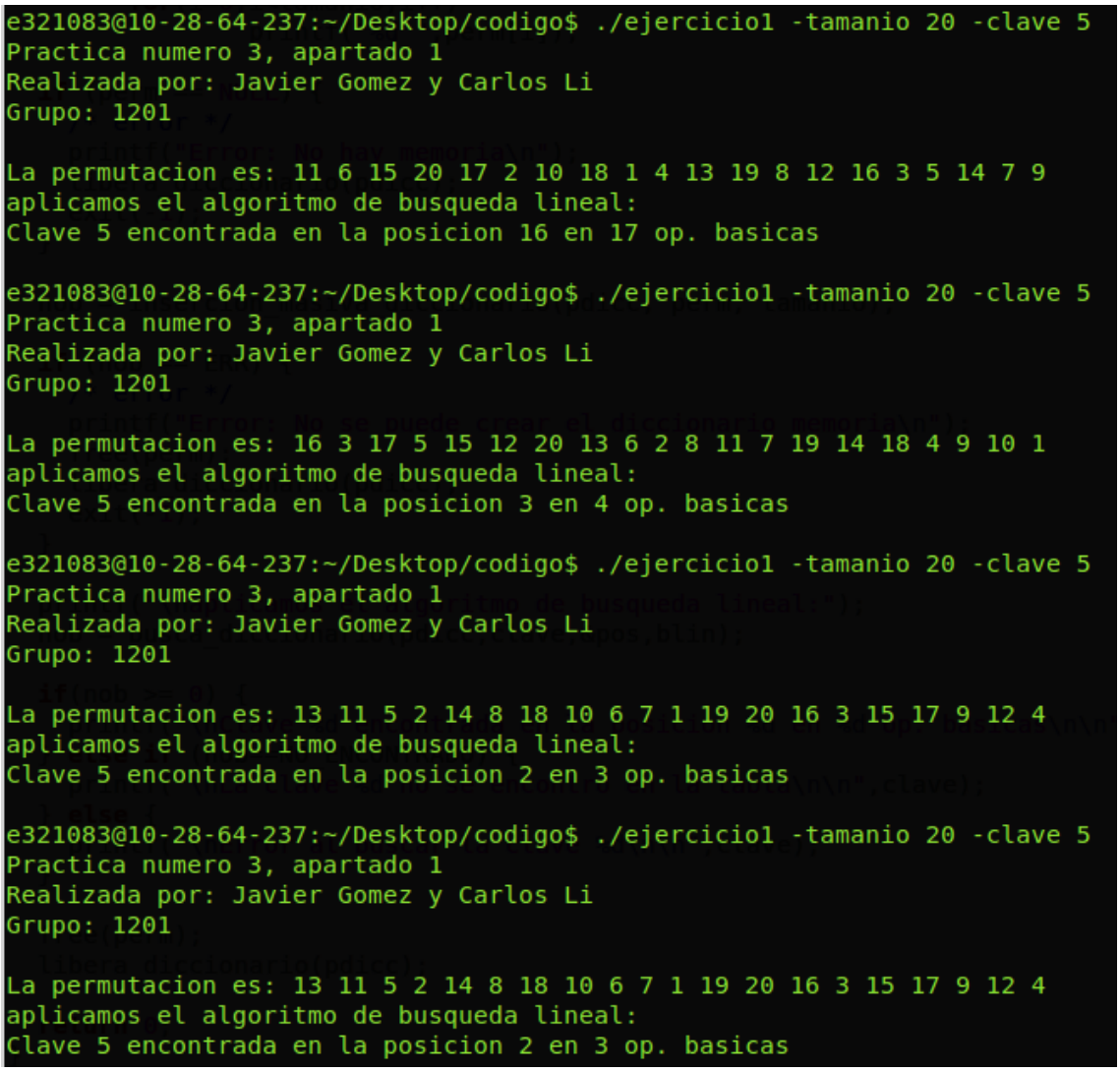
```
                clock_gettime(CLOCK_REALTIME, &time);


                t_medio += (time.tv_nsec - aux.tv_nsec) * 0.000000001;
                megamax += max[i];
                megamin += min[i];
                free(L[i]);
                libera_diccionario(dic);
        }
        free(L);
        megamax /= i;
        megamin /= i;
        ptiempo->n_perms = n_ciclos;
        ptiempo->tiempo = (double) t_medio / i;
   ptiempo->medio_ob = (double) OB / (i*k);
   ptiempo->max_ob = megamax;
   ptiempo->min_ob = megamin;
        ptiempo->tamanio = tamanio;
        free(min);
        free(max);
        free(claves);
   return OK;


}
```

5. Resultados, Gráficas

Aquí ponis los resultados obtenidos en cada apartado, incluyendo las posibles gráficas.

5.1 Apartado 1

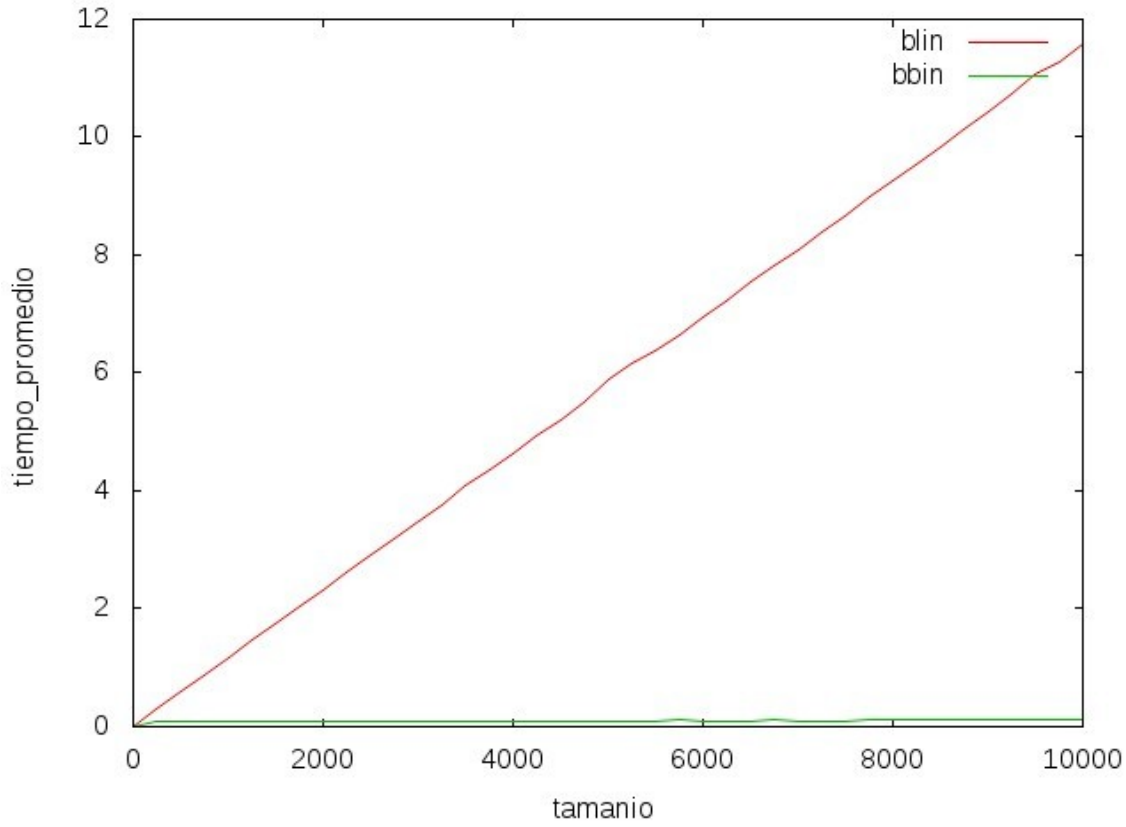
Adjuntamos imágenes de los resultados de los tres algoritmos estudiados:

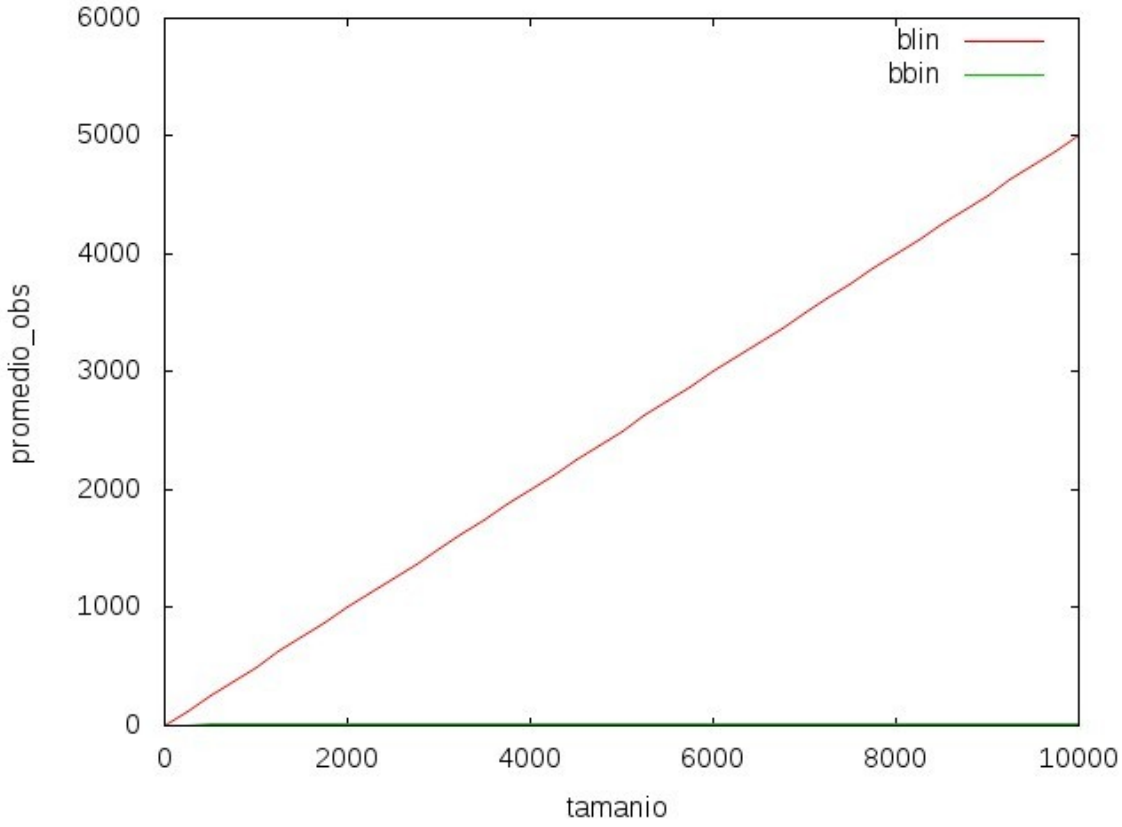


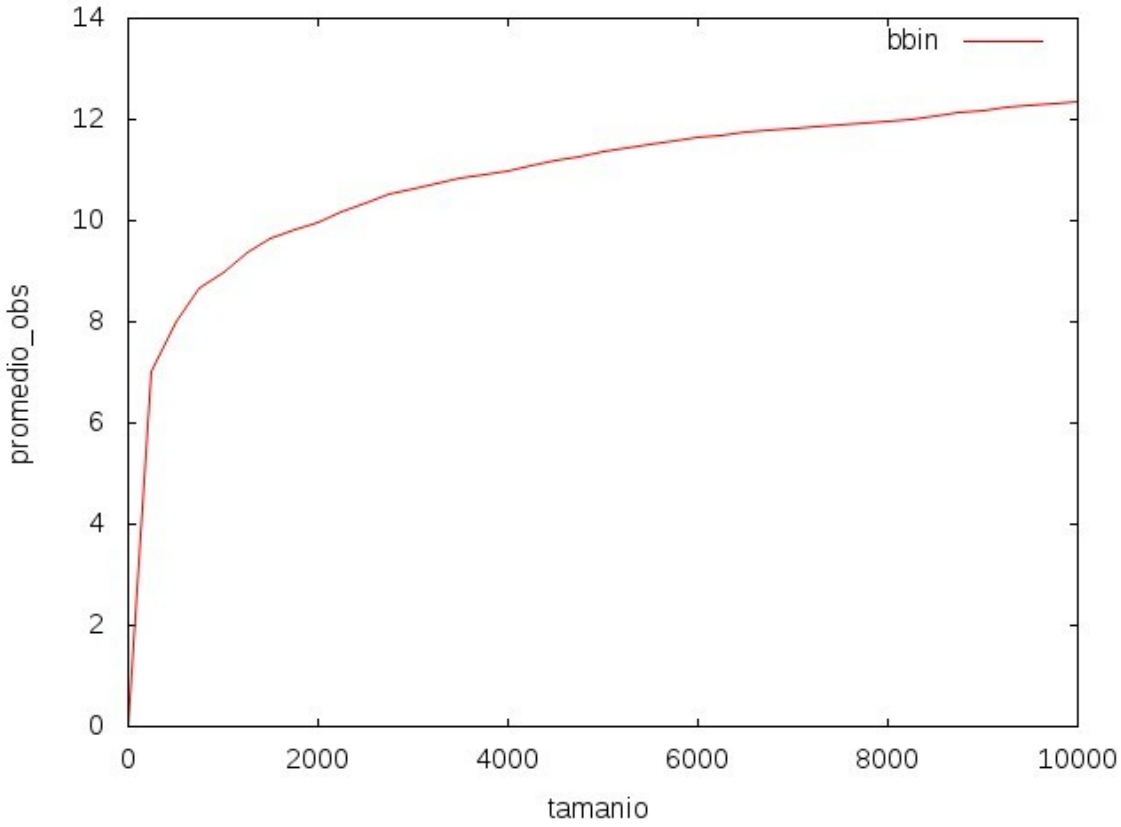
```
La permutacion es: 17 13 8 20 12 2 11 15 16 14 19 7 10 4 1 9 18 6 3 5
aplicamos el algoritmo de busqueda binaria:
Clave 5 encontrada en la posicion 4 en 2 op. basicas

e321083@10-28-64-237:~/Desktop/codigo$ ./ejercicio1 -tamanio 20 -clave 5
Practica numero 3, apartado 1
Realizada por: Javier Gomez y Carlos Li
Grupo: 1201

La permutacion es: 14 11 1 9 2 17 16 12 13 20 19 7 18 6 3 5 15 10 4 8
aplicamos el algoritmo de busqueda binaria:
Clave 5 encontrada en la posicion 4 en 2 op. basicas

e321083@10-28-64-237:~/Desktop/codigo$ ./ejercicio1 -tamanio 20 -clave 5
Practica numero 3, apartado 1
Realizada por: Javier Gomez y Carlos Li
Grupo: 1201

La permutacion es: 14 11 1 9 2 17 16 12 13 20 19 7 18 6 3 5 15 10 4 8
aplicamos el algoritmo de busqueda binaria:
Clave 5 encontrada en la posicion 4 en 2 op. basicas

e321083@10-28-64-237:~/Desktop/codigo$ ./ejercicio1 -tamanio 20 -clave 5
Practica numero 3, apartado 1
Realizada por: Javier Gomez y Carlos Li
Grupo: 1201

La permutacion es: 11 19 10 14 3 13 7 5 1 16 4 18 6 12 17 9 15 8 20 2
aplicamos el algoritmo de busqueda binaria:
Clave 5 encontrada en la posicion 4 en 2 op. basicas

e321083@10-28-64-237:~/Desktop/codigo$ ./ejercicio1 -tamanio 20 -clave 5
Practica numero 3, apartado 1
Realizada por: Javier Gomez y Carlos Li
Grupo: 1201

La permutacion es: 11 19 10 14 3 13 7 5 1 16 4 18 6 12 17 9 15 8 20 2
aplicamos el algoritmo de busqueda binaria:
Clave 5 encontrada en la posicion 4 en 2 op. basicas

e321083@10-28-64-237:~/Desktop/codigo$ ./ejercicio1 -tamanio 20 -clave 5
Practica numero 3, apartado 1
Realizada por: Javier Gomez y Carlos Li
Grupo: 1201

La permutacion es: 13 5 8 1 7 15 6 3 10 4 19 17 14 2 20 9 11 18 16 12
aplicamos el algoritmo de busqueda binaria:
Clave 5 encontrada en la posicion 4 en 2 op. basicas
```

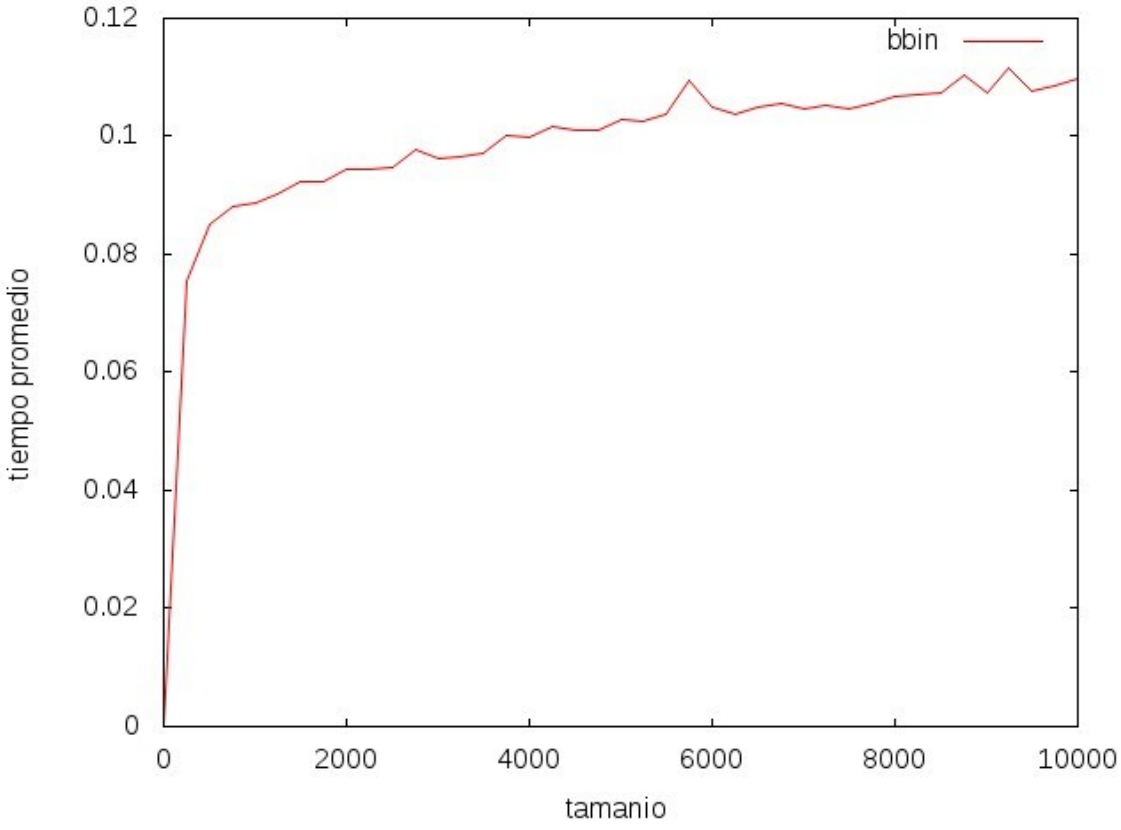
```
./ejercicio1 -tamanio <int> -clave <int>
Donde:
 -tamanio : numero elementos de la tabla.
 -clave : clave a buscar.
e321083@10-28-64-237:~/Desktop/codigo$ ./ejercicio1 -tamanio 20 -clave 5
Practica numero 3, apartado 1
Realizada por: Javier Gomez y Carlos Li
Grupo: 1201

La permutacion es: 11 12 14 6 2 18 13 1 4 7 9 20 8 19 10 15 17 16 3 5
aplicamos el algoritmo de busqueda lineal auto-organizada:
Clave 5 encontrada en la posicion 18 en 20 op. basicas

e321083@10-28-64-237:~/Desktop/codigo$ ./ejercicio1 -tamanio 20 -clave 5
Practica numero 3, apartado 1
Realizada por: Javier Gomez y Carlos Li
Grupo: 1201

La permutacion es: 18 20 19 12 3 14 11 15 10 5 1 17 13 2 8 6 4 9 7 16
aplicamos el algoritmo de busqueda lineal auto-organizada:
Clave 5 encontrada en la posicion 8 en 10 op. basicas

e321083@10-28-64-237:~/Desktop/codigo$ ./ejercicio1 -tamanio 20 -clave 5
Practica numero 3, apartado 1
Realizada por: Javier Gomez y Carlos Li
Grupo: 1201

La permutacion es: 15 18 6 17 10 4 9 2 20 1 13 14 16 3 5 7 8 19 11 12
aplicamos el algoritmo de busqueda lineal auto-organizada:
Clave 5 encontrada en la posicion 13 en 15 op. basicas

e321083@10-28-64-237:~/Desktop/codigo$ ./ejercicio1 -tamanio 20 -clave 5
Practica numero 3, apartado 1
Realizada por: Javier Gomez y Carlos Li
Grupo: 1201

La permutacion es: 15 18 6 17 10 4 9 2 20 1 13 14 16 3 5 7 8 19 11 12
aplicamos el algoritmo de busqueda lineal auto-organizada:
Clave 5 encontrada en la posicion 13 en 15 op. basicas

e321083@10-28-64-237:~/Desktop/codigo$ ./ejercicio1 -tamanio 20 -clave 5
Practica numero 3, apartado 1
Realizada por: Javier Gomez y Carlos Li
Grupo: 1201

La permutacion es: 19 5 8 20 16 9 12 13 6 2 3 17 18 14 15 1 7 11 4 10
aplicamos el algoritmo de busqueda lineal auto-organizada:
Clave 5 encontrada en la posicion 0 en 2 op. basicas
```

5.2 Apartado 2

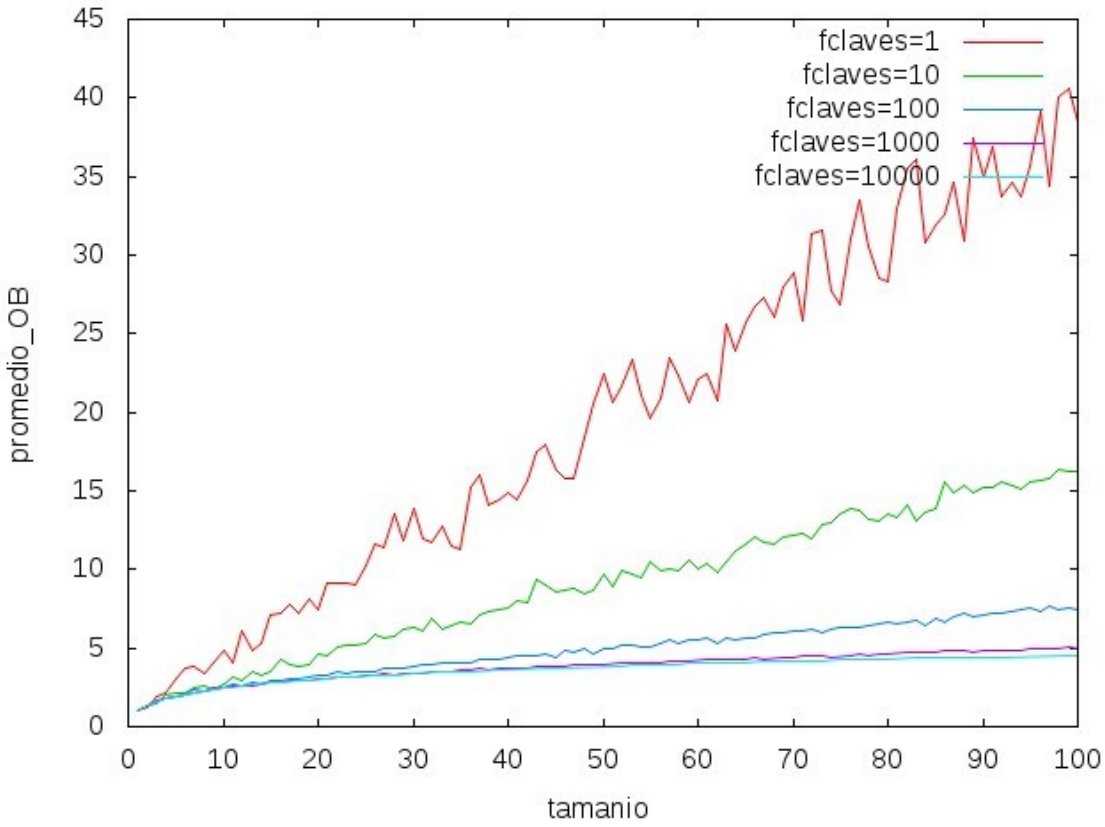


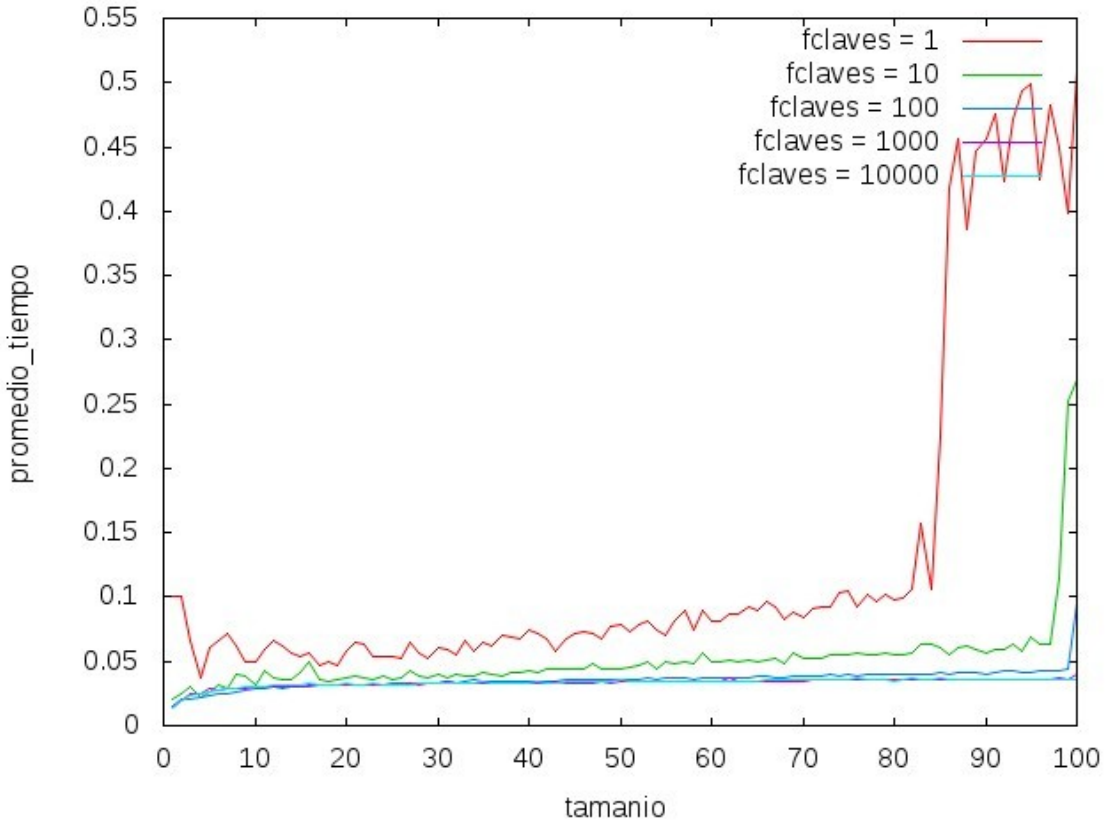



bbin
promedio_obs
tamanio
0
2
4
6
8
10
12
14
0
2000
4000
6000
8000
10000


bbin
tiempo promedio
tamanio
0
0.02
0.04
0.06
0.08
0.1
0.12
0
2000
4000
6000
8000
10000

Estas son las gráficas de comparación entre el promedio de operaciones básicas y promedio de tiempo de la búsqueda lineal y de la búsqueda binaria. Como podemos apreciar, la búsqueda lineal crece linealmente y es significativamente más grande que la búsqueda binaria, que crece logarítmicamente. Esto último tal vez no pueda apreciarse con demasiado detalle por estar el eje y en una escala en la que la búsqueda lineal quepa. Pero si podría verse en las últimas dos imágenes.





Estas gráficas contienen las comparaciones de las ejecuciones del algoritmo de búsqueda lineal auto-organizada con respecto del número promedio de Obs y el número promedio de tiempo. Como podemos observar, para valores de fclaves pequeños (como 1 10 o 100) el algoritmo es menos consistente tanto en tiempo de ejecución como en promedio de operaciones básicas. Esto es debido a que para valores más grandes de fclaves se buscan más veces las claves comunes y por tanto se tarda menos en encontrarlas más adelante. Por este mismo motivo, el crecimiento de las gráficas es aproximadamente logaritmico. A excepción del final, donde crece bastante debido al inmenso tamaño de la tabla de claves y en cierta medida, a la aleatoriedad de la permutación donde esté buscando.

5. Respuesta a las preguntas teóricas.

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

5.1 Pregunta 1

La OB de BBin es la comparación de clave `if (tabla[*ppos] == clave)`
Y sus else if consiguientes.

La OB de Blin y Blin_auto es la comparación de clave `if (clave == tabla[i])`

5.2 Pregunta 2

Wbb(N)=techo(logN) Bbb(N)=1

Wbl(N)=N Bbl(N)=1

5.3 Pregunta 3

Al emplear la búsqueda lineal auto-organizada sobre una tabla (no ordenada) buscando claves generadas de forma potencial, las claves más comunes se van posicionando cada vez más cerca del comienzo de la tabla y por tanto reduciendo el tiempo de ejecución de futuras búsquedas de la misma clave y mejorando el rendimiento del algoritmo.

Esto es porque el algoritmo de búsqueda lineal auto-organizada, al encontrar la clave que buscaba, intercambia su posición con la clave que hubiera en la posición anterior (si es que hay clave en dicha posición). Y dado que las claves son generadas de forma potencial, es decir, hay pocas claves con mucha frecuencia y muchas claves con poca frecuencia, en la tabla se busca reiteradas veces la misma clave, de manera que dicha clave se va acercando más al comienzo tal y como hemos enunciado al comienzo.

5.4 Pregunta 4

Dado que ya se han realizado un elevado número de búsquedas con búsqueda lineal auto-organizada y que la tabla está más o menos estable, el tiempo de ejecución medio de este algoritmo para buscar claves generadas de manera no uniforme sino potencial, es asintótico a O(1), pues probablemente la nueva clave a buscar sea una de las que ya están al principio de la tabla, que es el caso mejor, y aun de no ser así, se realizarían más búsquedas de claves como la que acabamos de comentar que de otras claves y por tanto el promedio seguiría teniendo valores tan pequeños como ese.

5.5 Pregunta 5

Suponiendo que todos los elementos de una tabla estén ordenados, el algoritmo de búsqueda lineal funcionará correctamente porque al partir una tabla escogiendo el elemento de la “mitad”, ésta se divide quedando por un lado todos los elementos menores al elemento escogido y por el otro lado con todos los elementos mayores al mismo. De esta forma, al comparar la clave que se busca con el elemento escogido, pueden suceder tres cosas: La primera de todas y el caso más sencillo, es que coincida que el elemento escogido y el elemento que se buscaba fueran el mismo, en cuyo caso ya se habría encontrado. Puede suceder también que el elemento sea o bien mayor o bien menor que el elemento a buscar, y dependiendo de esto, se descartan los elementos de la mitad de la lista que correspondan (si es mayor, se descartan todos los elementos menores y si es menor se descartan todos los elementos mayores) y se repite el proceso con la sublista resultante donde sí puede encontrarse la clave que se desea encontrar.

Esto es:

para una tabla X = [0, N] tal que para todo x(n) perteneciente a X sucede que x(n+1) > x o bien x(n+1) no existe y además que x(n-1) < x o bien x(n-1) no existe (en otras palabras, los elementos de X están ordenados).

Se desea encontrar un elemento k que existe en la tabla.

Siguiendo el algoritmo de búsqueda lineal, se escoge el elemento m = (0+N)/2 y se compara con k. En el caso de que k sea igual a m, se termina el algoritmo con éxito.

En caso de que k sea menor que m (no es restrictivo suponer esto por simetría de la demostración), sé que todo elemento x(n) perteneciente a X tal que n >= m cumple que x(n) > k y por tanto defino una nueva tabla Y = [0, m] cuyos elementos sean iguales a los m primeros elementos de X y que además conserven también el orden. Esta tabla contiene a k (puesto que la tabla X contenía a k y todos los elementos desde la posición m en adelante son mayores y por tanto distintos de k) . Sobre esta nueva tabla, vuelvo a aplicar el algoritmo de búsqueda binaria.

Se reitera el algoritmo hasta que, en caso de no haberse podido encontrar en ninguna de las iteraciones anteriores, se llegue a una tabla K cuyo único elemento será necesariamente k, ya que en cada una de las subtablas que definíamos en cada iteración estábamos asegurando que k estuviera en ellas.

6. Conclusiones finales.

En esta práctica hemos cumplido con los objetivos de entender el funcionamiento de los algoritmos de búsqueda con sus ventajas y desventajas, así como cuándo resultan más efectivos.

Para empezar, la búsqueda lineal es un método bastante sencillo de entender y por decirlo de alguna forma, estándar. Debido a que funciona comparando uno a uno los elementos de la tabla con el que se esta buscando hasta que lo encuentre, el rendimiento de este algoritmo es independiente a cualquier condición impuesta, es decir, no requiere que la tabla esté ordenada ni que las claves a buscar sean generadas de una manera en específico, pues siempre va a mantener su rendimiento más o menos estable (dependiendo, claro está, de la aleatoriedad de la permutación). Sin embargo, aun siendo el más versátil de los tres para cualquier situación, también es el que tiene el peor rendimiento, dado que su tiempo de ejecución es lineal y depende del tamaño de la tabla. En caso de que no estén seguras ninguna condición de las tablas o de las claves a buscar, o que no tengan ninguna propiedad en concreto, el mejor algoritmo es éste.

El algoritmo de búsqueda binaria resulta ser el más eficaz de los tres en lo que a tiempo de ejecución se refiere, pues no se comporta de forma lineal sino logarítmica, lo cual para tablas muy grandes es mucha diferencia. Sin embargo, tiene el inconveniente de que exige que la tabla esté ordenada, pues de no ser así, no se garantiza que el elemento a buscar se encuentre (de hecho en la mayoría de los casos no se encuentra). En el caso de que la tabla esté ordenada, ésta es indiscutiblemente la mejor opción para buscar elementos mediante comparaciones de claves, tanto es así que hasta puede resultar rentable el ordenar la tabla.

Finalmente, el algoritmo de búsqueda lineal auto-organizada es un algoritmo similar al de búsqueda lineal con una pequeña particularidad, ya comentada en apartados anteriores, que hace que éste sea una buena elección si, teniendo una tabla desordenada, las claves no se generan de forma uniforme, es decir, si ciertas búsquedas son más frecuentes que el resto. Esto es debido a que va acercando al comienzo de la tabla las claves que vaya encontrando, quedando las más comunes al principio de todo. Por tanto, en el caso de que se cumpla esta condición pero no se cumpla que la tabla esté

ordenada, éste algoritmo es la mejor elección. Como desventaja tiene que, de aplicarse en una tabla ya ordenada, esta tabla quedaría desordenada, siendo el precio a pagar tal vez demasiado alto pues no permitiría hacer búsquedas binarias más tarde. Además, dada una distribución uniforme de claves, este algoritmo puede resultar peor que el algoritmo de búsqueda lineal pero bajo ninguna circunstancia mejor, también por el hecho de que desordene un poco la tabla en cada búsqueda.