Análisis de Algoritmos 2021/2022

Práctica 3

Pablo Almarza, Grupo.

1. Introduction

In this practice functions to run a a dictionary data structure will be created. Also, three different functions will be created to search for a key and another two to check the efficiency of the algorithms.

2. Objectives

2.1 Section 1

In this first section the functions for the dictionary and the searching functions will be created. After that, they will be checked with the exercise1.c file.

2.2 Section 2

In this second section the two functions to test the searching algorithms will be implemented and will be corrected with the exercise2.c file.

3. Tools and methodology

The program used is Visual Studio on Linux. Using C/C++ for VS and Valgrind when executing the programs to check errors and memory leaks

3.1 Section 1

For the dictionary data structure it was implemented following the instructions in the comments on the functions which we had to create. For the search algorithms the explanation given in class was followed.

3.2 Section 2

For the two new functions, the idea was taking the first functions created on times.c and adapt them in order to implement the key generator and the dictionary with the code given.

4. Source code

4.1 Section 1

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

 PDICT pdict=NULL;

 if(size < 1 || (order != SORTED && order != NOT_SORTED)) {

   return NULL;

 }

  pdict=(PDICT)malloc(sizeof(DICT));

 if(!pdict) {

   return NULL;

 }

  pdict->table = NULL;

  pdict->table = (int*)malloc(sizeof(int)*size);

 if(!pdict->table) {

   free_dictionary(pdict);

   return NULL;

 }

 pdict->size = size;

 pdict->n_data = 0;

 pdict->order = order;

  return pdict;

}
```

```
void free_dictionary(PDICT pdict)
{
 if(!pdict) return;
 if(pdict->table!=NULL) {
   free(pdict->table);
 }
 free(pdict);
}
int insert_dictionary(PDICT pdict, int key)
{
 int A, j = 0, count = 0;
 if(!pdict || pdict->size == pdict->n_data) return ERR;
 pdict->table[pdict->n_data] = key;
  if(pdict->order==SORTED && pdict->n_data > 0){
   A = pdict->table[pdict->n_data];
   j = pdict->n_data - 1;
   while (j >= 0 && pdict->table[j] > A){
     pdict->table[j+1] = pdict->table[j];
     j--;
     count++;
   }
   if(j >= 0){
     count++;

   }

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

 }

 pdict->n_data++;

 return count;

}

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

{

 int count = 0, i, aux;



 if(!pdict || !keys || n_keys<1 || (pdict->size - pdict->n_data) <
n_keys) return ERR;

  for(i = 0 ; i < n_keys ; i++){

   if((aux = insert_dictionary(pdict, keys[i])) == ERR) {

     return ERR;

   }

   count += aux;

 }

 return count;

}

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

```
{
 if(!pdict || !method || !ppos || key < 0) return ERR;
 return method(pdict->table, 0, pdict->n_data-1, key, ppos);
}
/* Search functions of the Dictionary ADT */
int bin_search(int *table,int F,int L,int key, int *ppos)
{
 int M, count = 0;
 if(!table || F > L || F < 0) return ERR;
 while(F <= L){
   M = (L+F)/2;
   count++;
   if(table[M] == key){
     *ppos = M;
     return count;
   }
   else if(table[M] > key){
     L = M-1;
   }
   else{
     F = M+1;
   }
 }

 *ppos = NOT_FOUND;

 return count;

}

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

{

 int count = 0, i;

  if(!table || F > L || F < 0 || !ppos) return ERR;

  for(i=F ; i <= L ; i++){

   count++;

   if(table[i] == key){

     *ppos=i;

     return count;

   }

 }

 *ppos = NOT_FOUND;

 return count;

}

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

{

 int count = 0, i;

 if(!table || F > L || F < 0 || !ppos) return ERR;
```

```
  for(i = F ; i <= L ; i++){

    count++;

    if(table[i] == key){

      if(i != F){

        swap(&table[i], &table[i-1]);

        i--;

      }

      *ppos=i;

      return count;

    }

  }

  *ppos = NOT_FOUND;

  return count;

}
```

4.2 Section 2

```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 n = 0, i = 0, j = 0;

 if(!method || !generator || !file || incr <= 0 || num_min >
num_max || num_min < 0 || (order != NOT_SORTED && order != SORTED)
|| n_times <= 0){

   return ERR;

 }

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

 ptime = malloc(sizeof(TIME_AA) * n);

 if(!ptime) return ERR;

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

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

     free(ptime);

     return ERR;

   }

 }
```

```c
  if(save_time_table(file, ptime, n) == ERR){

   free(ptime);

   return ERR;

 }

  free(ptime);

 return OK;

}

short average_search_time(pfunc_search metodo, pfunc_key_generator
generator,

                             int order,

                             int N,

                             int n_times,

                             PTIME_AA ptime)

{

 int i=0, *perm=NULL, *keys=NULL, ppos=0, aux=0, n_ob=0;

 clock_t begin, end;

 double time;

 PDICT pdict=NULL;



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

  ptime->average_ob = 0;

 ptime->max_ob = 0;
```

```
ptime->min_ob = 0;

ptime->time = 0;



pdict = init_dictionary(N, order);

if(!pdict) {

  return ERR;}



perm = generate_perm(N);

if(!perm){

  free_dictionary(pdict);

  return ERR;

}



if(massive_insertion_dictionary(pdict, perm, N) == ERR){

  free_dictionary(pdict);

  free(perm);

  return ERR;

}



keys = (int*)malloc(sizeof(int) * N * n_times);

if(!keys){

  free_dictionary(pdict);
  free(perm);

  return ERR;

}

generator(keys, n_times * N, N);



begin=clock();

for(i = 0; i < N * n_times; i++){

  aux = search_dictionary(pdict, keys[i], &ppos, metodo);

  if(aux == ERR || ppos == NOT_FOUND){

    free_dictionary(pdict);

    free(perm);

    free(keys);

    return ERR;

  }



  if(aux < ptime->min_ob || ptime->min_ob == 0) {

    ptime->min_ob = aux;

  }



  if(aux>ptime->max_ob || ptime->max_ob == 0) {

    ptime->max_ob = aux;

  }
```

```
   n_ob+=aux;

 }

 end=clock();



 time = (double)(end-begin)/CLOCKS_PER_SEC;

 time = time/(N*n_times);



 ptime->time = time;

 ptime->average_ob = (double)(n_ob)/(N*n_times);

 ptime->N = N;

 ptime->n_elems = N * n_times;



 free_dictionary(pdict);

 free(perm);

 free(keys);

  return OK;

}
```

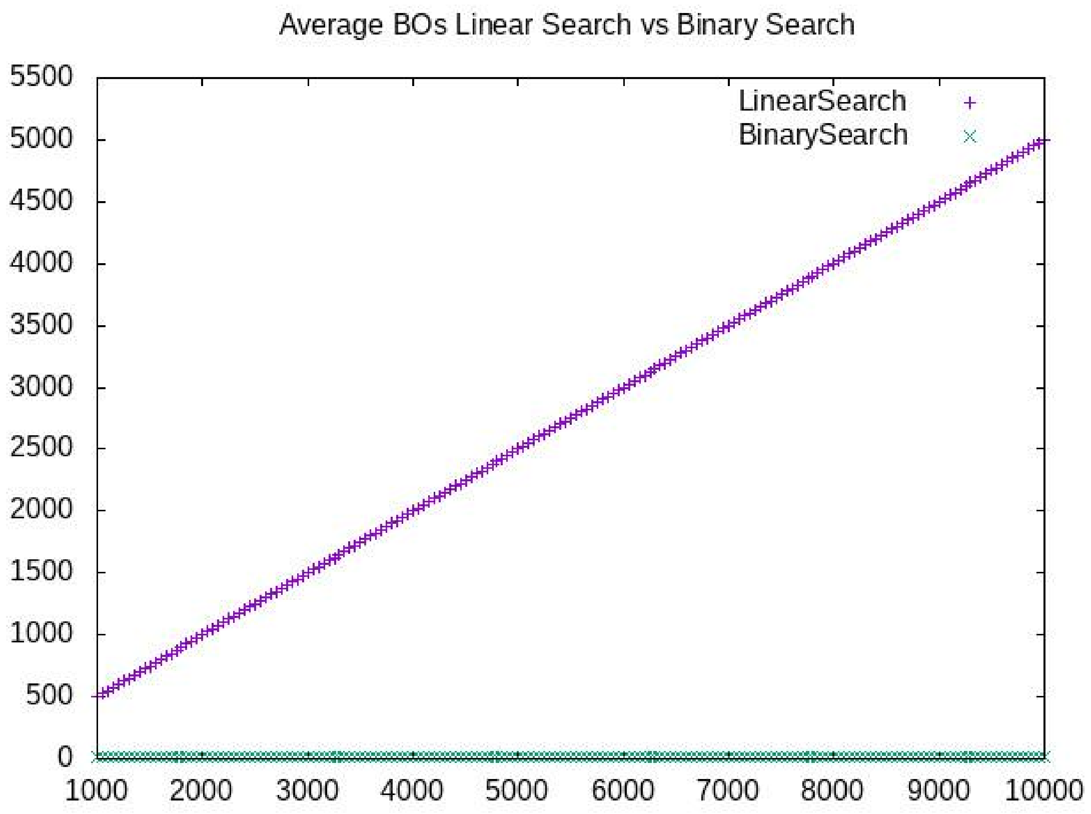
5. Results, Plots

5.1 Section 1

```
./exercise1 -size 10 -key 4

Pratice number 3, section 1

Done by: Pablo Almarza

Group: Your group

Key 4 found in position 3 in 4 basic op.
```

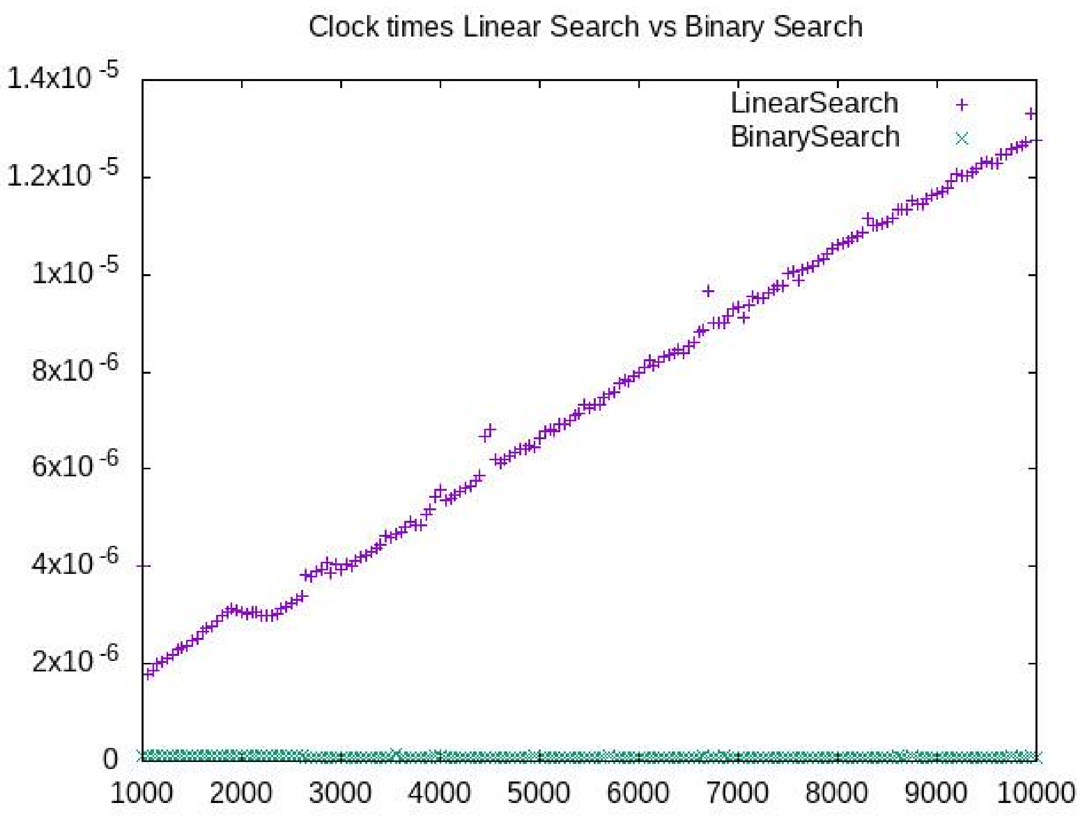
5.2 Section 2

Plot comparing the average number of OBs between linear search (not sorted) and binary search (sorted), comments to the graph.



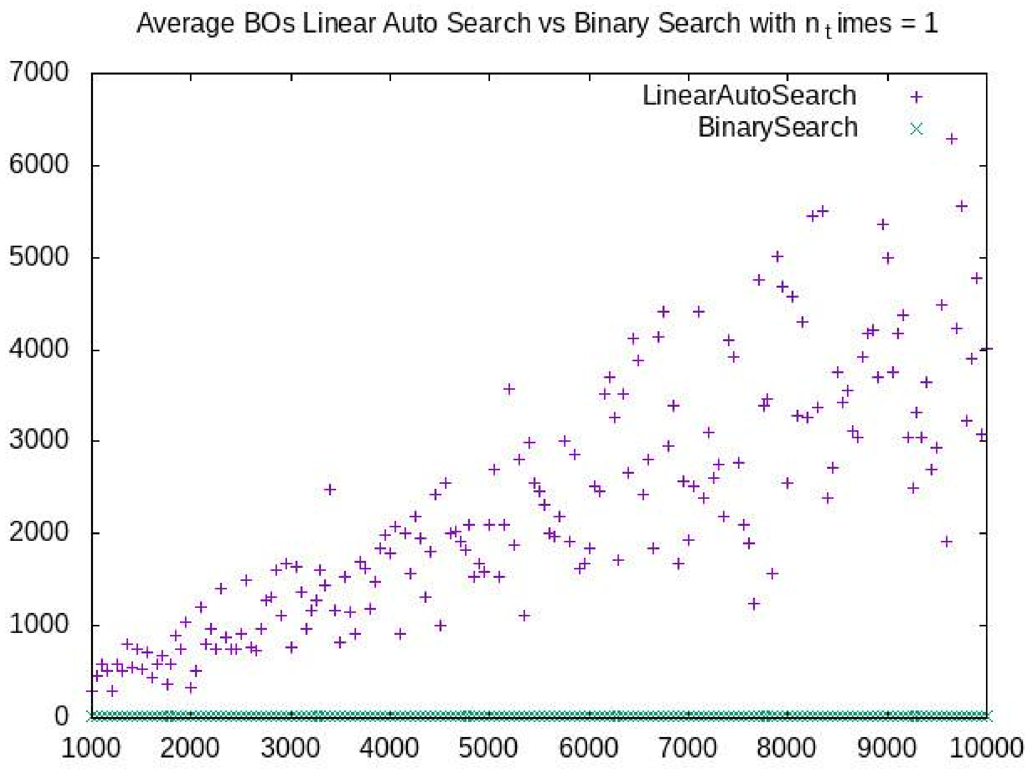
We can observe that linear search needs a lot more basic operations. That is because with uniform key generation Linear Search average case obs is (N+1)/2 while average case for Binary Search is $\log_2 N$.

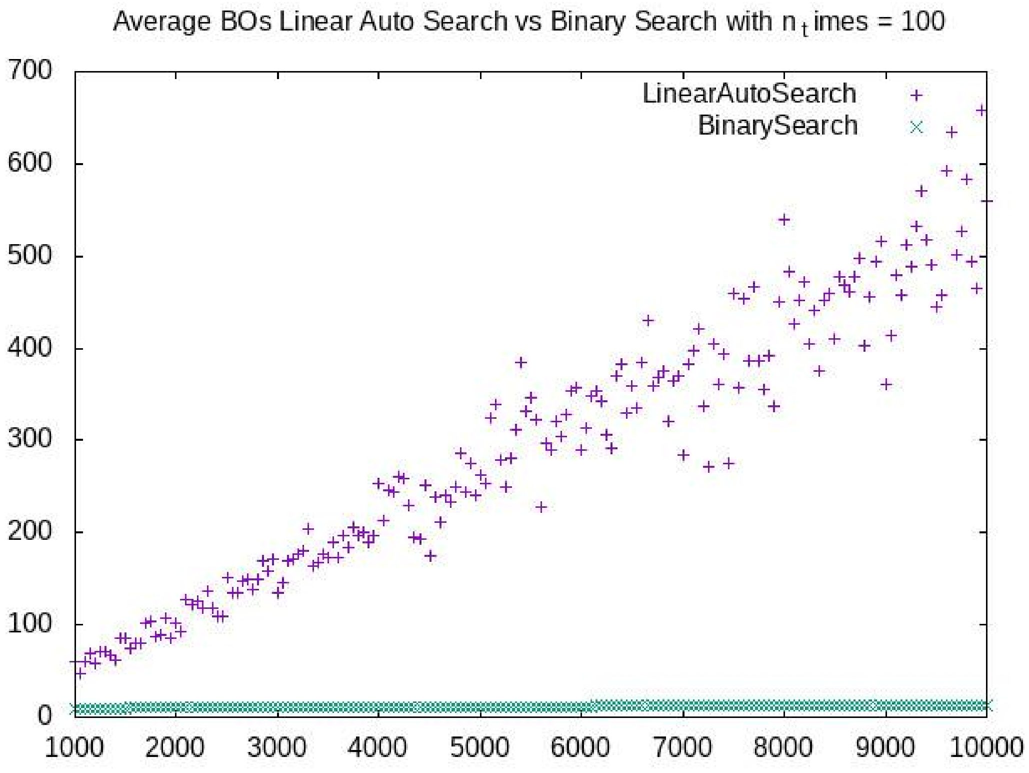
Plot comparing the average clock time between the linear search and the binary search, comments to the graph.

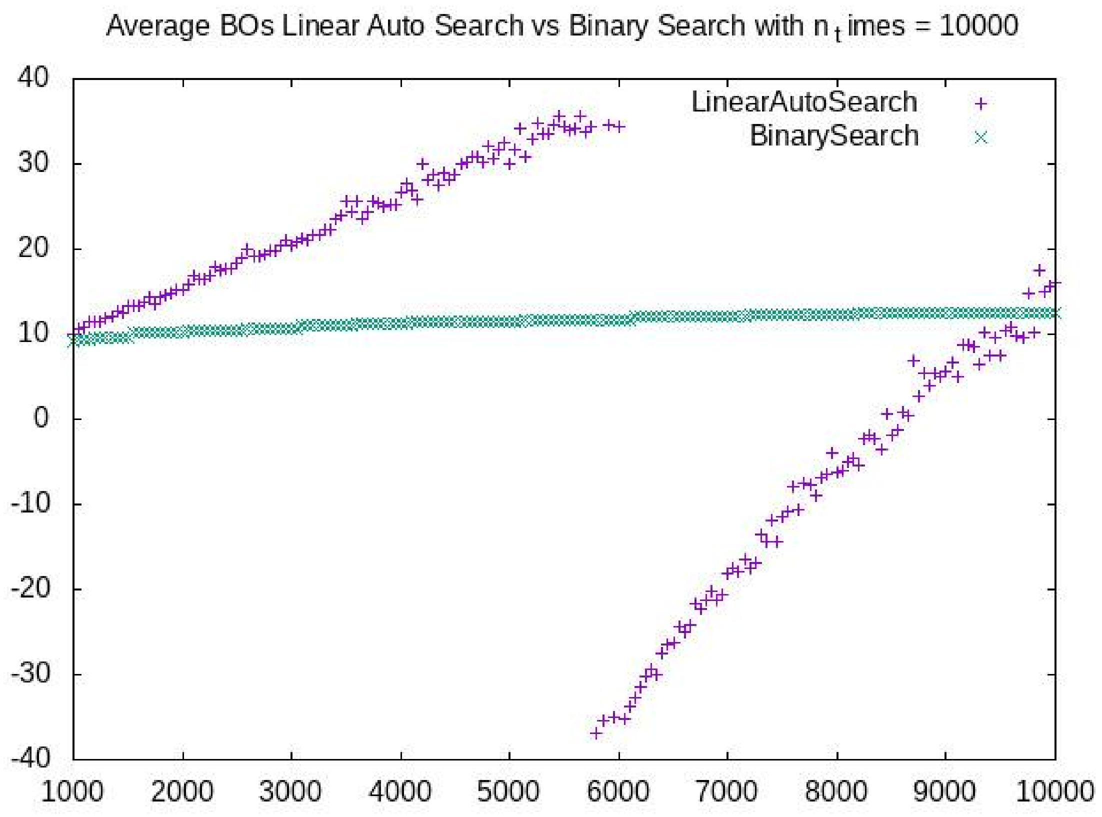


As a result of the difference between basic operations, Binary Search is much faster than Linear Search.

Plot comparing the average number of OBs between the binary search and the self-organized linear search (for the values of n_times = 1, 100 and 10000), comments to the graph.

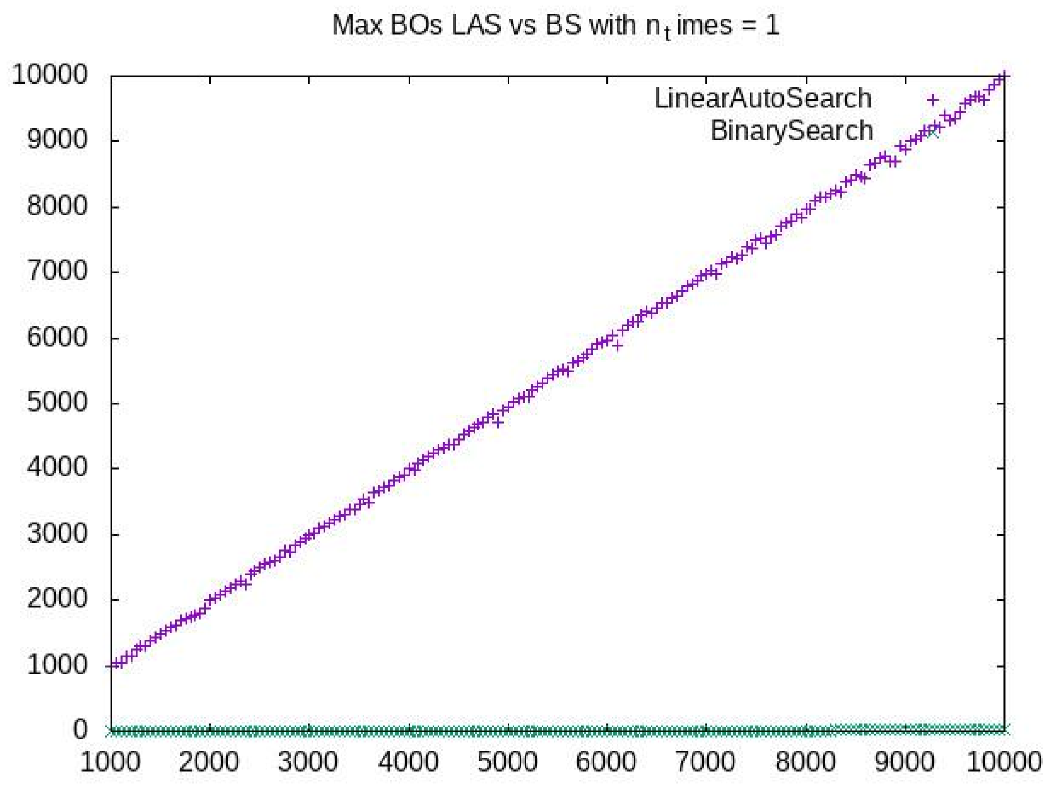


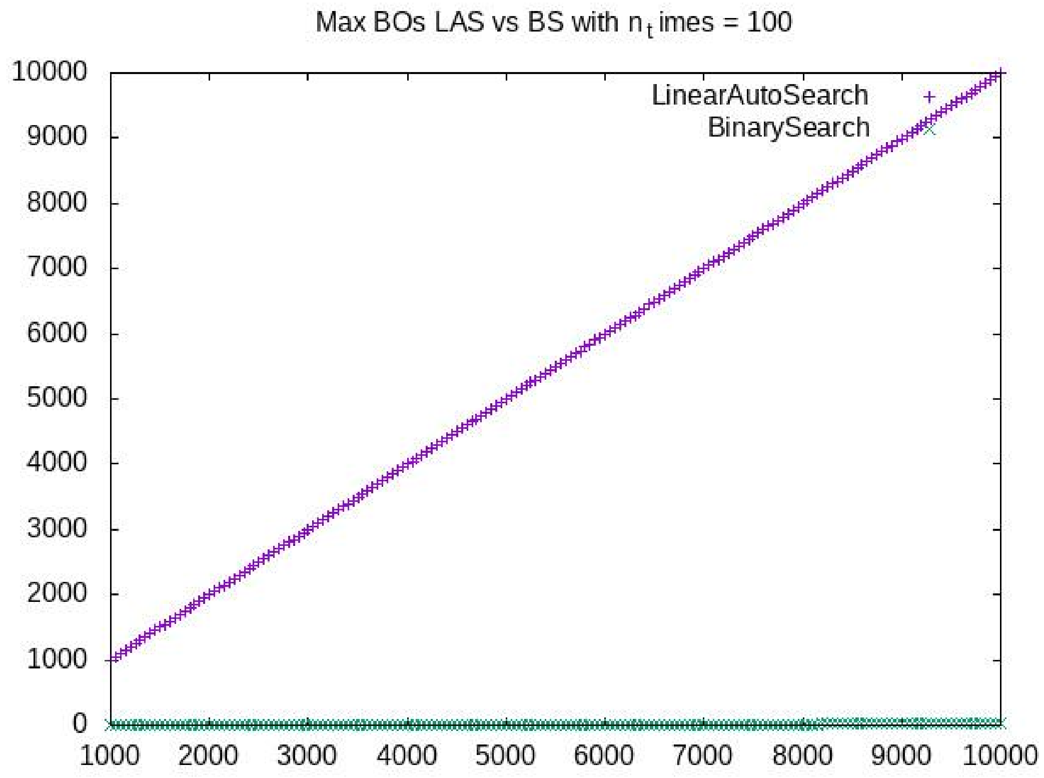


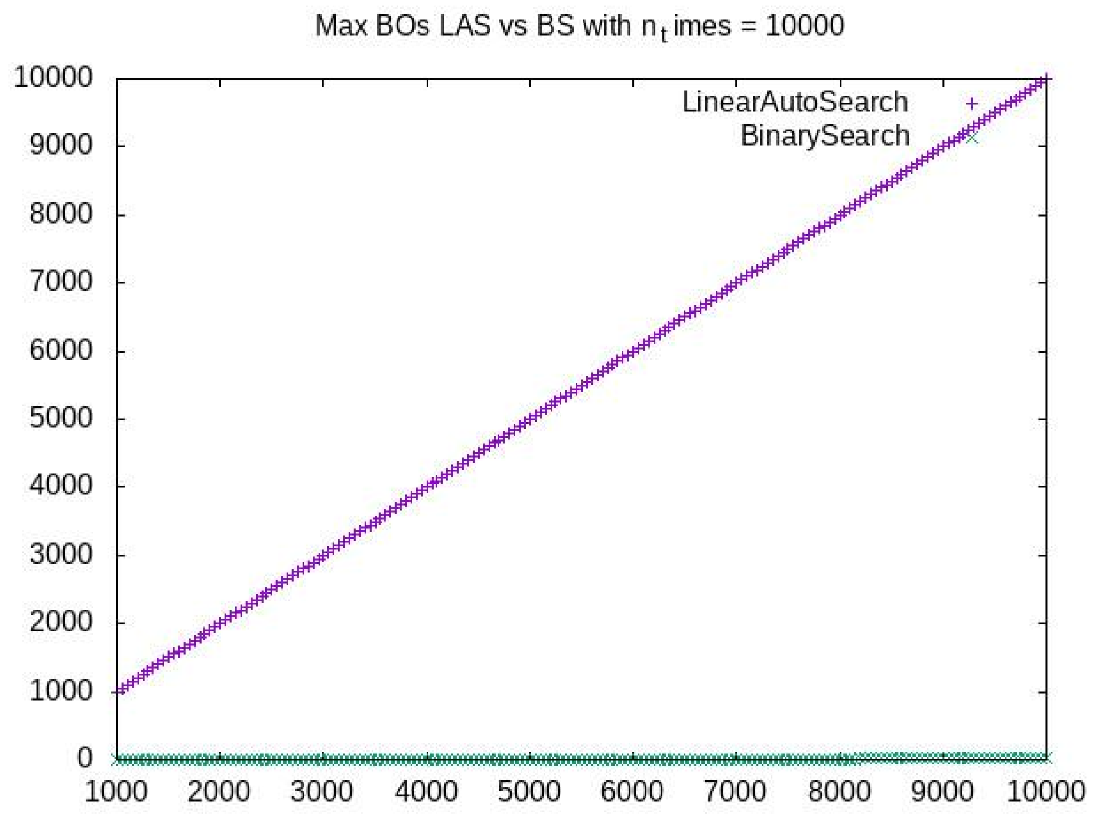


It can be seen that the more n_times, the more efficient linear auto search is but it is not enough to be better than binary search. To be honest, I don't know what happens with n_times = 10000, something in the code happens that starts taking negative numbers but, if you get the inverse points, the graphic of the linear auto search is almost a straight line.

Plot comparing the maximum number of OBs between the binary search and the self-organized linear search (for the values of n_times = 1, 100 and 10000), comments to the graph.

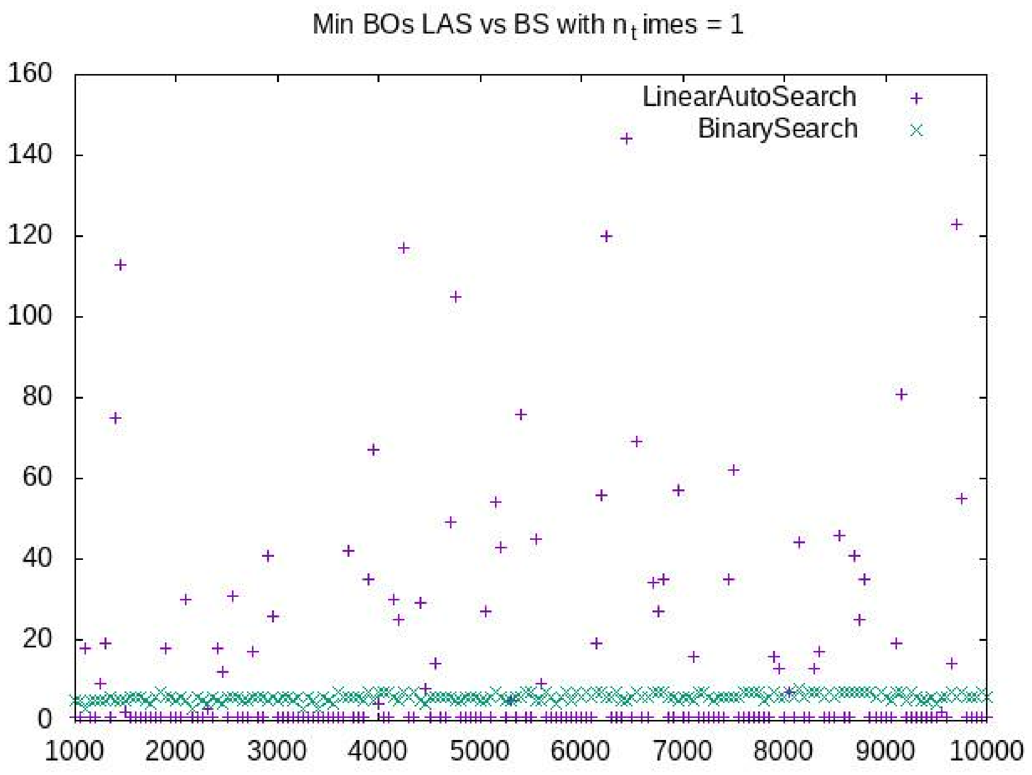


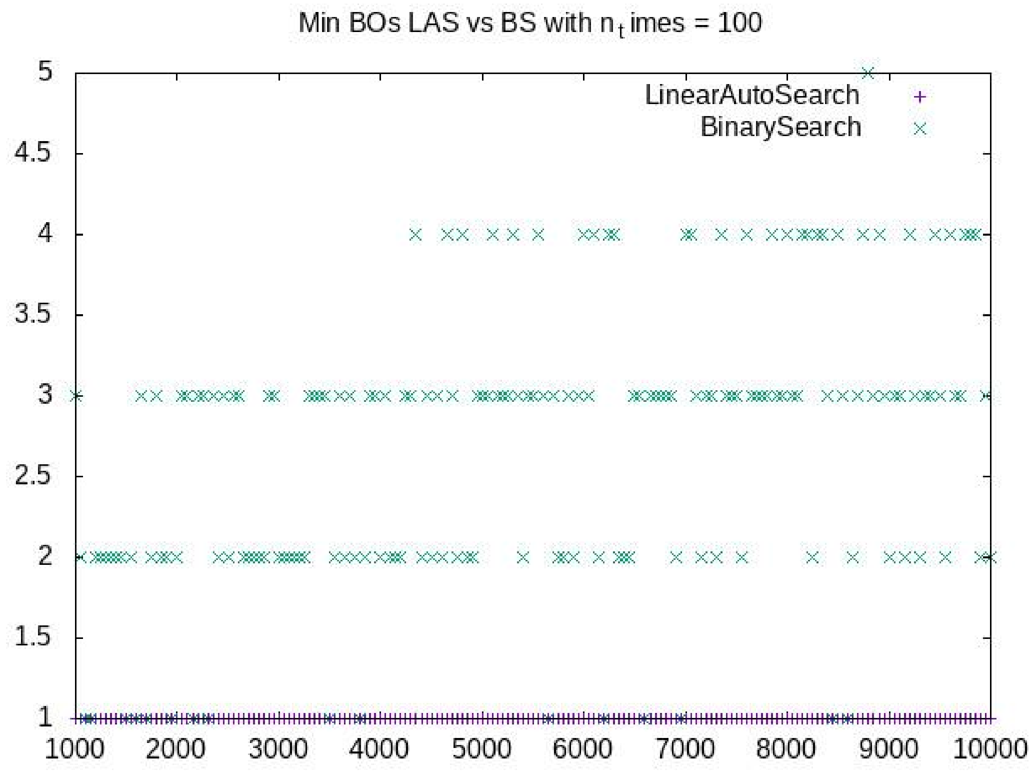


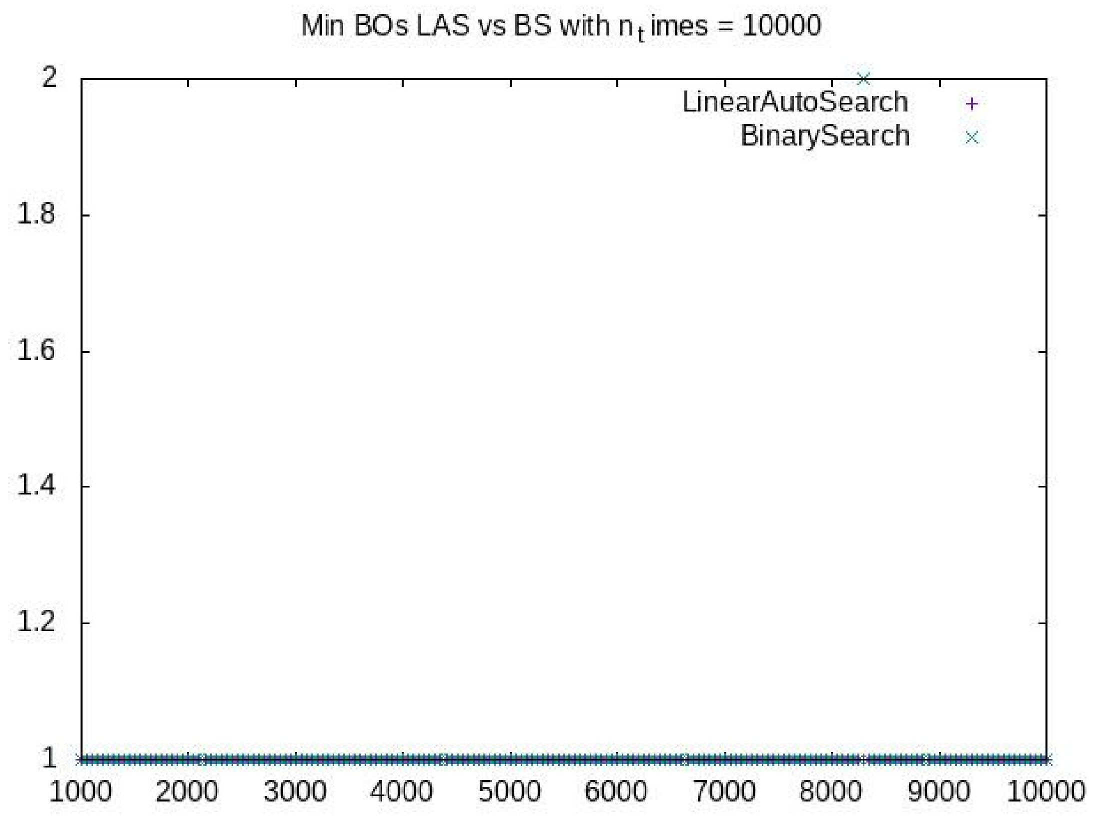


In these plots we can see that linear auto search is worse than binary search in the worst case (when the number of BOs is max). Therefore we can say that binary search is better than linear auto search.

Plot comparing the minimum number of OBs between the binary search and the self-organized linear search (for the values of n_times = 1, 100 and 10000), comments to the graph.

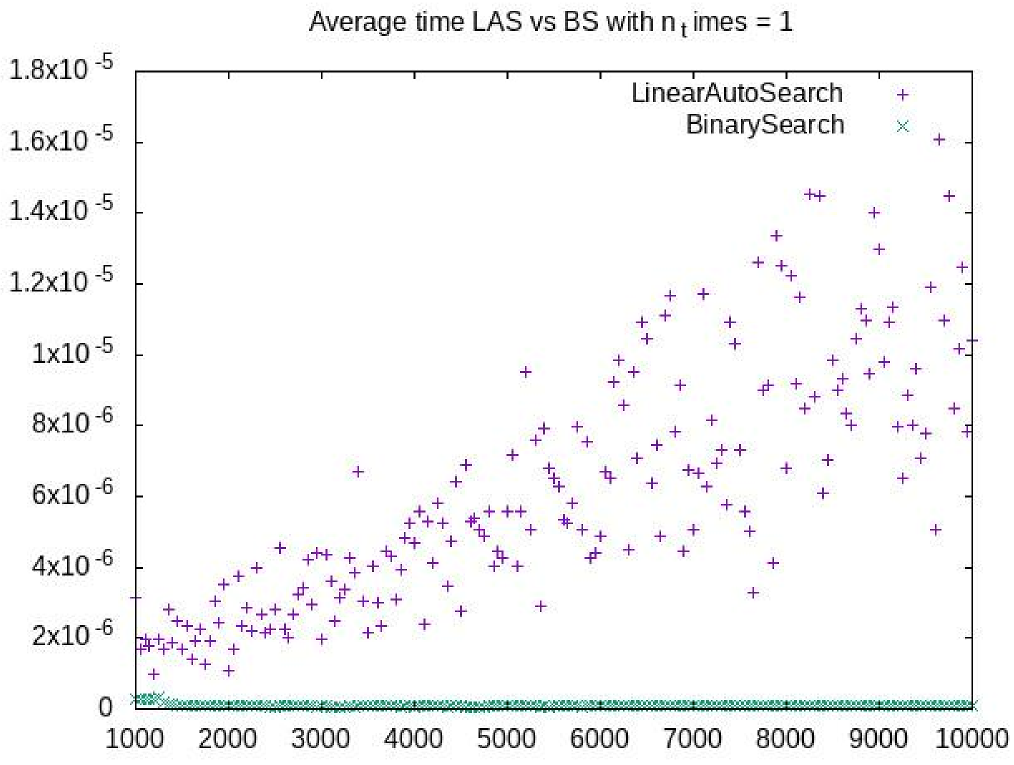


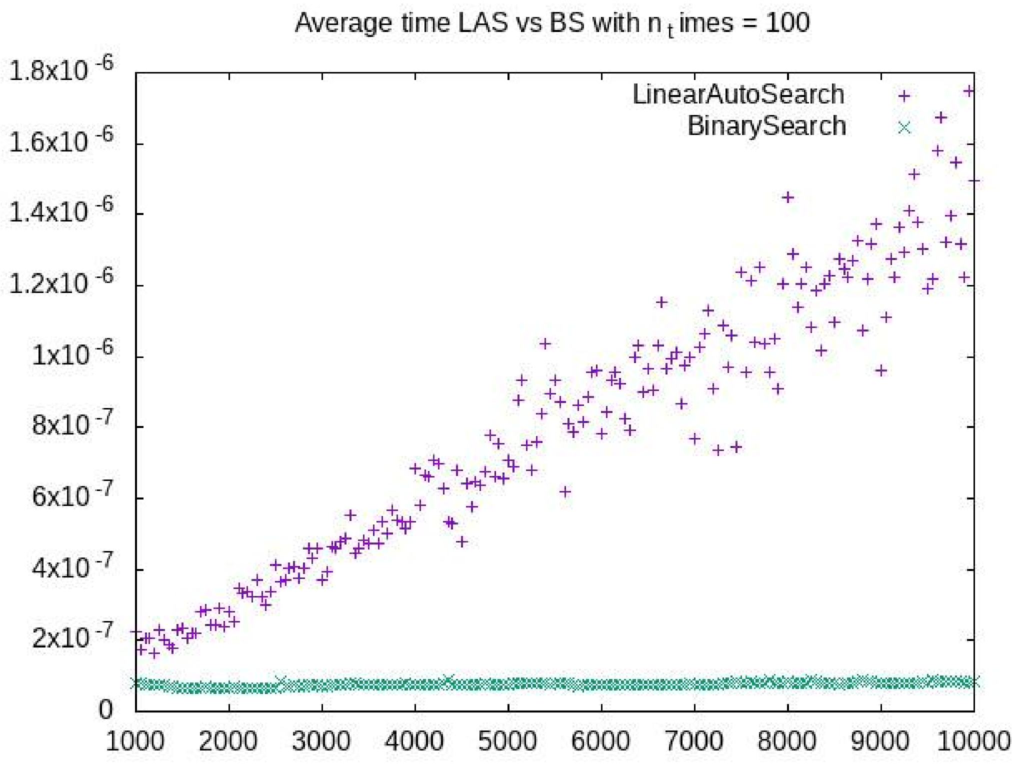


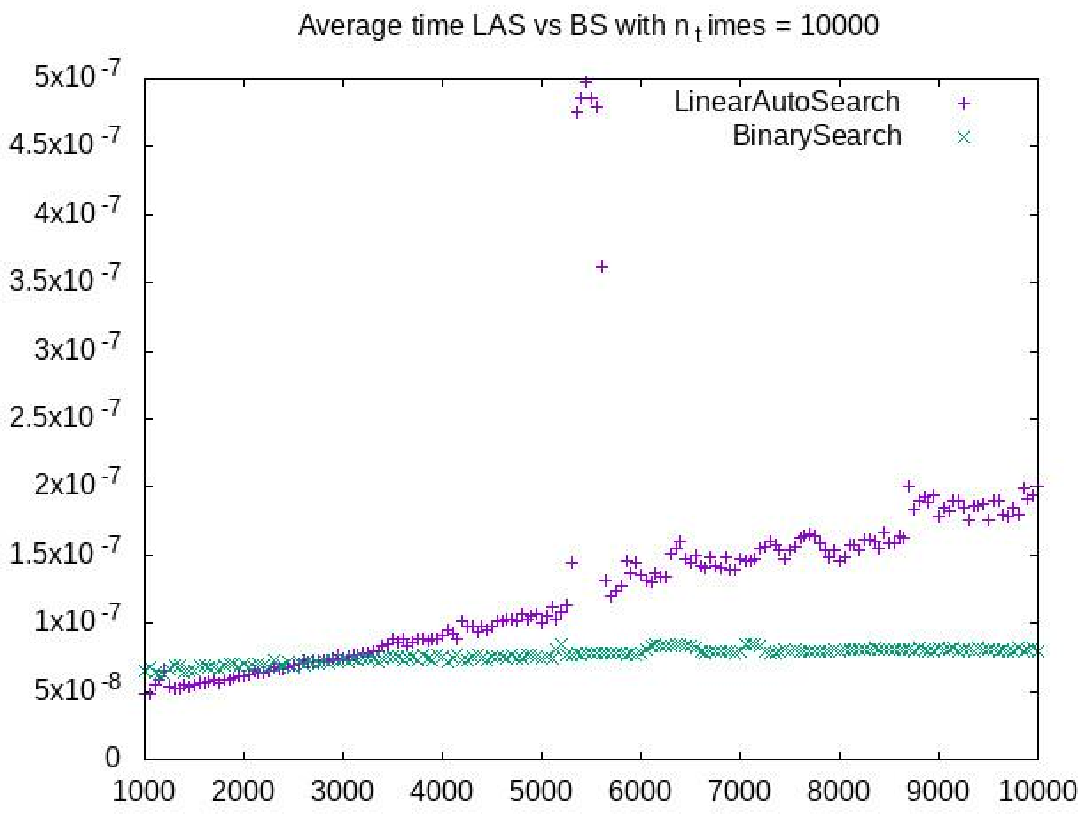


Comparing these plots we can see that linear auto search is usually better because, since we use the potential key generator, values that are repeated are more likely to be in the first positions of the array. In the binary search case, we will usually have a second search in order to explore the tree.

Plot comparing the average time between the binary search and the self-organized linear search (for the values of n_times = 1, 100 and 10000), comments to the graph.







It can be seen that the more n_times, the more a straight line the graph for linear auto search is, but while linear auto search with n_times very big is better at some point, binary search is better on average so it can be said that binary search is better in general.

5. Answers to theoretical Questions.

5.1 Pregunta 1

BO_{lin_search}= table[i] == key

BO_{bin_search}= table[M] == key

$BO_{lin\_auto\_search}$= table[i] == key

5.2 Pregunta 2

$W_{BS}(N)= \Omega(\log(N))$

$B_{BS}(N)= O(1)$

$W_{LS}(N) = N$

$B_{LS}(N) = O(1)$

5.3 Pregunta 3

As the number of searches increases, the numbers that are found more times go to the first indexes of the table so it reduces the time that the algorithm takes to find them.

5.4 Pregunta 4

It is of the order of O(NlogN).

5.5 Pregunta 5

Binary search orders the array so an element has two children. The left child has less value than the parent and the right child is greater than his parent. So, when searching, it will compare the key value with the element and if it is smaller it will follow the left way and, if it is greater, it will follow the right way.

6. Final Conclusions.

As a conclusion, three new search algorithms have been implemented and, comparing them, binary search is the most efficient.