Analysis of Algorithms 2019/2020

Practice 3

Rodrigo Juez and José Manuel Freire, Group 1291.

Code	Plots	Documentation	Total

1. Introduction.

In this practice, we had to create a dictionary, where we store numbers in its table. After that we had to create 3 searching algorithms: linear search, auto-organized linear search and binary search, and then measure its efficiency.

We will use the knowledge from the theory class to implement the search algorithms from the exercises so we can see the real results of the BOs that we had to calculate before. Thanks to this, we will also see in which cases it is better to use each algorithm.

2. Objectives

The objectives of each section are:

2.1 Section 1

In the first session we have to create and ADT for the dictionary, and all the functions related to it. The ADT has:

- size: the size of the table.
- n_data: the number of ints stored in the table.
- order: boolean that shows if the table is sorted.
- table: the table with all the data.

And the functions are:

- `init_dictionary`: it creates the dictionary ADT and initializes it.
- `insert_dictionary`: it adds the element to the table, taking care if it is ordered or not to keep the order.
- `massive_insertion_dictionary`: it inserts an array of ints, calling insert_dictionary several times.
- `search_dictionary:` it looks for they key in the dictionary given with pdict using the search algorithm given by method.
- `free_dictionary:` It frees the memory allocated by the dictionary.
- `bin_search, lin_search and lin_auto_search:` 3 different searching algorithms to find a key in the dictionary.

2.2 Section 2

For the section 2, we must create the function average_search_time, which gives in the ptime argument a structure of the type TIME_AA. This structure contains:

– The size of the dictionary in the N field.

– The number of searched keys in the n elems field.

– The average execution time in the time field.

– The average number of times that the BO was executed in the avg ob field.

– The minimum number of times that the BO was executed min ob.

– The maximum number of times that the BO was executed max ob.

We also have to create average_search_time function which adds with massive_insertion_dictionary N elements. After this we must allocate memory for n_times*N keys and start searching them with the keys generated. It must count the BOs and the clock time and save the results on the ptime structure.

3. Tools and methodology

The environments, programs and software we used to get to the final results of our program in each section were:

3.1 Section 1

For the first section, we used visual studio code on windows 10 to write all the code of this section and the Linux subsystem for windows to execute it. We also used Valgrind in this step. After this, we tried the program on Ubuntu in the labs to be sure it worked properly.
To coordinate our work, we used Discord to talk while we programmed and visual studio code's live share to be able to modify the files at the same time.
We also used GitHub to have our code up to date all the time and for it be accessible from everywhere.

3.2 Section 2

For this second section, we used visual studio code on windows 10 again to write all the code and the Linux subsystem of windows for running it and Valgrind to check any possible hidden error. After this, we tried the program on Ubuntu in the labs to be sure it worked properly.
To coordinate our work we used visual studio code's live share to be able to modify the files at the same time.
We also used GitHub to have our code up to date all the time and for it be accessible from everywhere.

4. Source code

Here we include the source code for the routines we have developed for each exercise.

4.1 Section 1

```
PDICT init_dictionary (int size, char order)
{
  PDICT dict=NULL;
  if(size<=0 ||(order!=NOT_SORTED && order!=SORTED)){
    return NULL;
  }

  dict = (PDICT) malloc (sizeof(DICT));
  if(!dict) return NULL;
  dict->table = NULL;
  dict->table = (int*) malloc (size * sizeof(int));
  if(!dict->table) {
      free(dict);
      return NULL;
    }
  dict->order = order;
  dict->size = size;
  dict->n_data = 0;

  return dict;
}

/*typedef struct dictionary {
  int size;
  int n_data;
  char order;
  int *table;
} DICT, *PDICT;*/

void free_dictionary(PDICT pdict)
{
  if(!pdict) return;

  if(pdict->table) free((pdict->table));
  free(pdict);

}

int insert_dictionary(PDICT pdict, int key)
{ int i=0, OBs=1; /*this is for being able to have it in the while statem
ent we correct it at the end with a -1*/
  if(!pdict) return ERR;
  pdict->table[pdict->n_data] = key;
```

```
  pdict->n_data++;

  if(pdict->order == NOT_SORTED) return 0;
  if(pdict->order == SORTED) {
    i=pdict->n_data -2;
    while(i>=0    &&   pdict->table[i] > key && OBs++){
      pdict->table[i+1]=pdict->table[i];
      i--;
    }
    pdict->table[i+1]=key;
    return --OBs;
  }

  return ERR;
}

int massive_insertion_dictionary (PDICT pdict,int *keys, int n_keys)
{
  int i, OBs=0, buffer=0;
  if(!pdict||!keys||n_keys<1){
    return ERR;
  }
  for(i=0; i<n_keys; i++) {
    buffer = insert_dictionary(pdict, keys[i]);
    if(buffer == ERR) return ERR;
    OBs += buffer;
  }

  return OBs;

}


int search_dictionary(PDICT pdict, int key, int *ppos, pfunc_search metho
d)
{
  if(!pdict||!method||!ppos) return ERR;

  if(method==bin_search && pdict->order == NOT_SORTED){
    *ppos=ERR;
    return ERR;
  }

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


}
```

```c
/* Search functions of the Dictionary ADT only works with the table sorte
d*/
int bin_search(int *table,int F,int L,int key, int *ppos)
{ int m=0, BOs=0;
  if(!table||F>L||!ppos){
    return ERR;
  }
  while (F<=L){
    BOs++;
    m=(F+L)/2;
    if(table[m]==key) {*ppos=++m; return BOs;}
    else if (key < table[m]) L=m-1;
    else F=m+1;

  }
  *ppos = NOT_FOUND;
  return NOT_FOUND;
}

int lin_search(int *table,int F,int L,int key, int *ppos)
{
  int i, BOs=0, flag;

  if(!table||F>L||!ppos){
    return ERR;
  }

  for(i=F, flag=0; i<=L && flag==0; i++){

    BOs++;
    if(table[i]==key){
      flag=1;
    }
  }

  if(flag==0) {*ppos = NOT_FOUND;
               return NOT_FOUND;}
  else *ppos = i;

  return BOs;
}

int lin_auto_search(int *table,int F,int L,int key, int *ppos)
{
  int i, BOs=0, flag, aux;

  if(!table||F>L||!ppos){
    return ERR;
  }
```

```
  for(i=F, flag=0; i<=L && flag==0; i++){

    BOs++;
    if(table[i]==key){
      if(i==0){*ppos=++i; return BOs;}
      flag=1;
    }
  }
  i--; /*real index*/

  if(flag==0) {*ppos = NOT_FOUND;
               return NOT_FOUND;}
  else {
    aux=table[i-1];
    table[i-1]=table[i];
    table[i]=aux;
    *ppos = i;} /*I return i and not i-
1 becuase whe need to return it user friendly which would be the new inde
x +1: i-1+1=i*/

  return BOs;
}
```

4.2 Section 2

```
short generate_search_times(pfunc_search method, pfunc_key_generator gene
rator,
                            int order, char* file,
                            int num_min, int num_max,
                            int incr, int n_times)
{
  int i=0, arraylenth=0, size_permutation=0;
  PTIME_AA times_array = NULL;

  if(!method || !file || num_min < 1 || num_max < num_min || incr < 0 ||
n_times < 1 || !generator) return ERR;

  if(incr != 0) for(i=num_min; i<=num_max; i+=incr) arraylenth++;
  else arraylenth = num_max - num_min + 1;

  size_permutation = num_min;

  times_array = (PTIME_AA)malloc(sizeof(TIME_AA)*arraylenth);
  if(!times_array) return ERR;
  /*INITIALIZE ARRAY*/
  for(i=0; i<arraylenth; i++){
```

```
    times_array[i].N = 0;
    times_array[i].n_elems = 0;
    times_array[i].time = 0;
    times_array[i].average_ob = 0;
    times_array[i].max_ob = 0;
    times_array[i].min_ob = MAX_INT;
  }

  for(i=0; i<arraylenth; i++){
    if(size_permutation > num_max){
      free(times_array);
      times_array = NULL;
      return ERR;
    }

    if(average_search_time(method, generator, order, size_permutation, n_
times, &(times_array[i])) == ERR) {
      free(times_array);
      return ERR;
    };
    size_permutation += incr;

  }


  save_time_table(file, times_array, arraylenth);


  free(times_array);
  return OK;
}









short average_search_time(pfunc_search method, pfunc_key_generator genera
tor,
                         int order,
                         int N,
                         int n_times,
                         PTIME_AA ptime)
{
  PDICT pdict = NULL;
  int* perm = NULL;
```

```
  int* keys = NULL;
  struct timespec start, end;
  long seconds, nanoseconds;
  double time=0;
  int  bufferOBs = 0, i, ppos = 0;
  if(!method || !generator || (order != SORTED && order != NOT_SORTED) ||
 N < 0 || n_times < 0 || !ptime) return ERR;
  /*PTIME INIT*/
    ptime->max_ob = 0;
    ptime->min_ob = MAX_INT;
    ptime->average_ob = 0;
    ptime->N = N;
    ptime->n_elems = N*n_times;

  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_times*N);
  if(!keys){
    free_dictionary(pdict);
    free(perm);
    return ERR;
  }

  generator(keys, N*n_times, N);

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

    if(clock_gettime(CLOCK_REALTIME, &start)==ERR){
      printf("\nerror in clock function at start\n");
      free_dictionary(pdict); free(perm); free(keys);
      return ERR;
    }
    if(keys[i] > N){printf("el potential keys generator devuelve numeros
mas grandes que el N que le damos");}
    bufferOBs = search_dictionary(pdict, keys[i], &ppos, method);
    if(ppos < 0 || bufferOBs < 0){
```

```
      printf("\nkey %i wasn't found\n", keys[i]);
      free_dictionary(pdict); free(perm); free(keys);
      return ERR;
    }
    if(clock_gettime(CLOCK_REALTIME, &end)==ERR){
      printf("\nerror in clock function at end\n");
      free_dictionary(pdict); free(perm); free(keys);
      return ERR;
    }
    ptime->average_ob += bufferOBs;
    if(ptime->max_ob < bufferOBs) ptime->max_ob = bufferOBs;
    if(ptime->min_ob > bufferOBs) ptime->min_ob = bufferOBs;
    seconds = end.tv_sec - start.tv_sec;
    nanoseconds = end.tv_nsec - start.tv_nsec;

    if (start.tv_nsec > end.tv_nsec) {
      --seconds;
      nanoseconds += 1000000000;
    }
    time+=seconds;
    time+=(float)nanoseconds/BILLION;


  }
  time = time/(double)(N*n_times);
  ptime->time = time;
  ptime->average_ob = ptime->average_ob/(N*n_times);
  free(perm); free(keys);
  free_dictionary(pdict);
  return OK;
}
```

5. Results, plots

Results obtained for the different exercises, and their plots.

5.1 Section 1

All of them were executed with

```
./exercise1 -size 100 -key 8
```

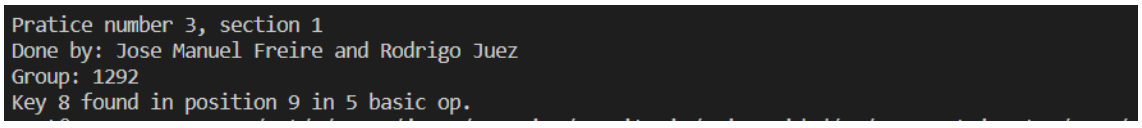
Linear Search NOT_SORTED:

Binary Search NOT_SORTED:

This error is expected to happen, because bin_search only works with ordered dictionaries.

Linear Search SORTED:

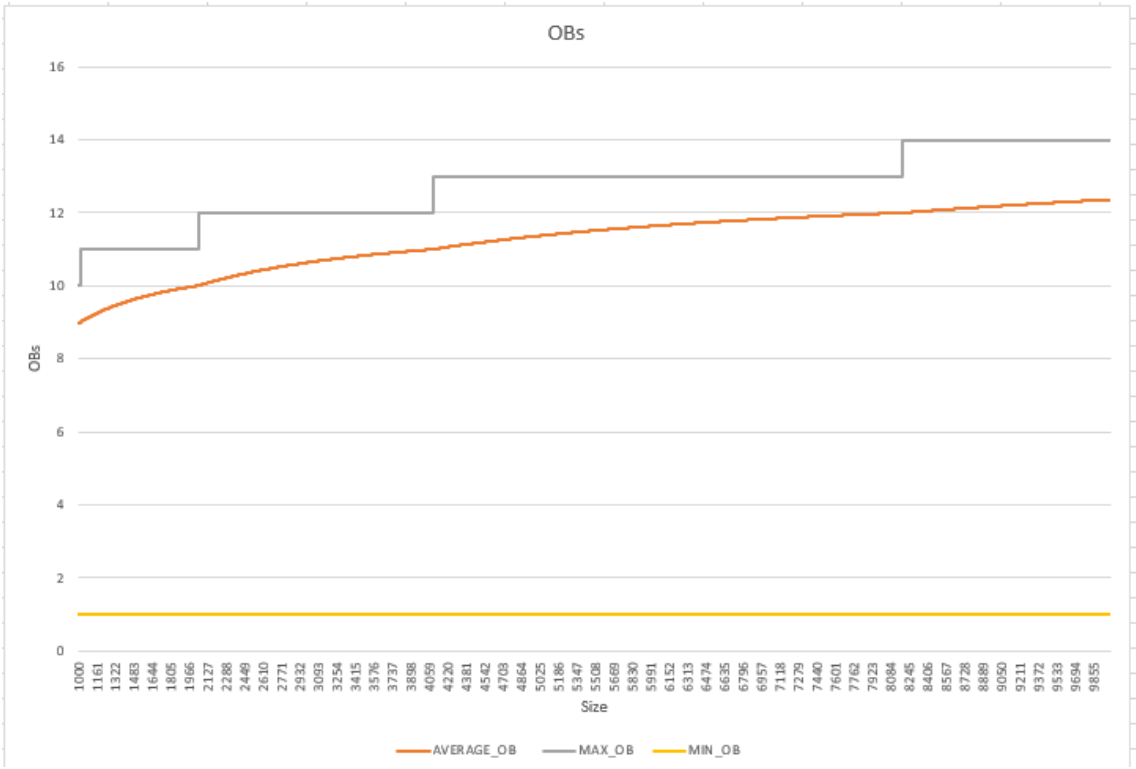
Binary Search SORTED:



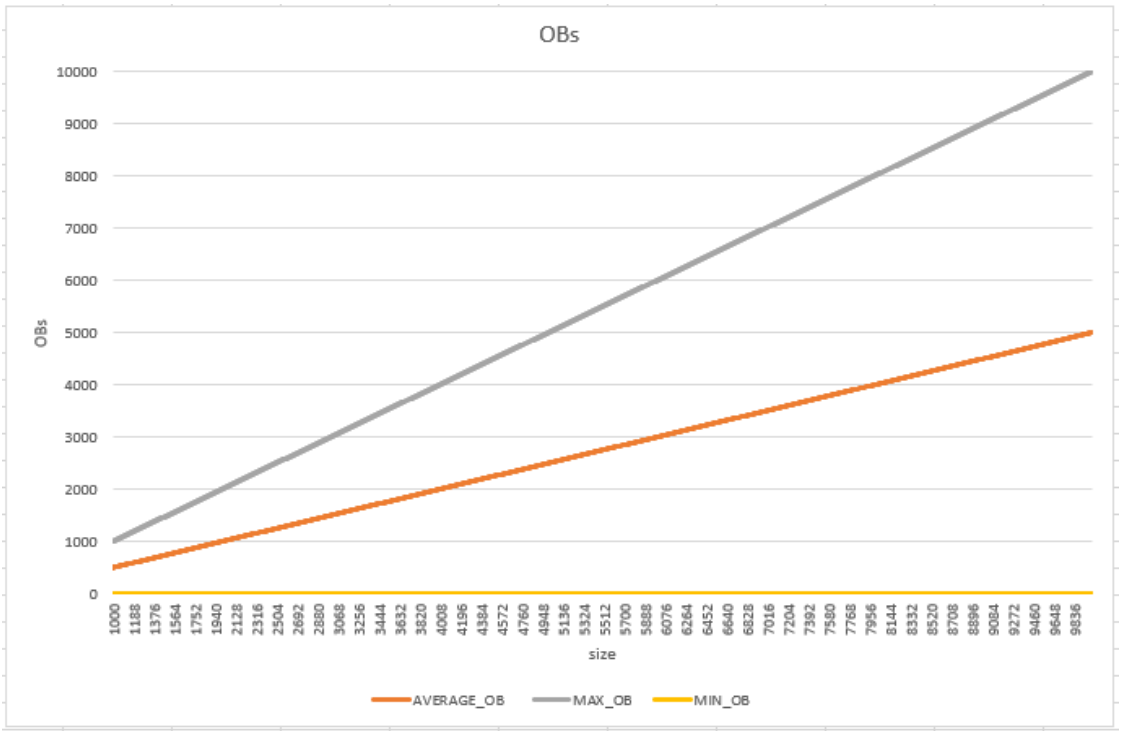
5.2 Section 2

Plot comparing the average number of BOs of linear and binary search approaches, comments to the plot.

Binary Search:

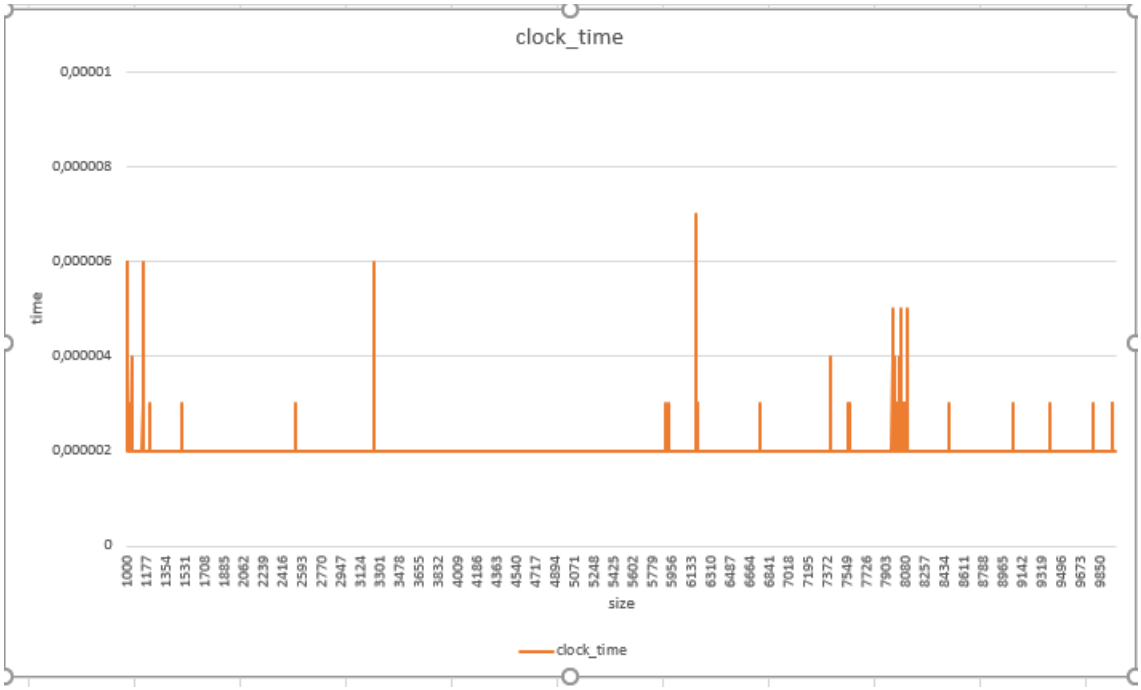


Linear Search:

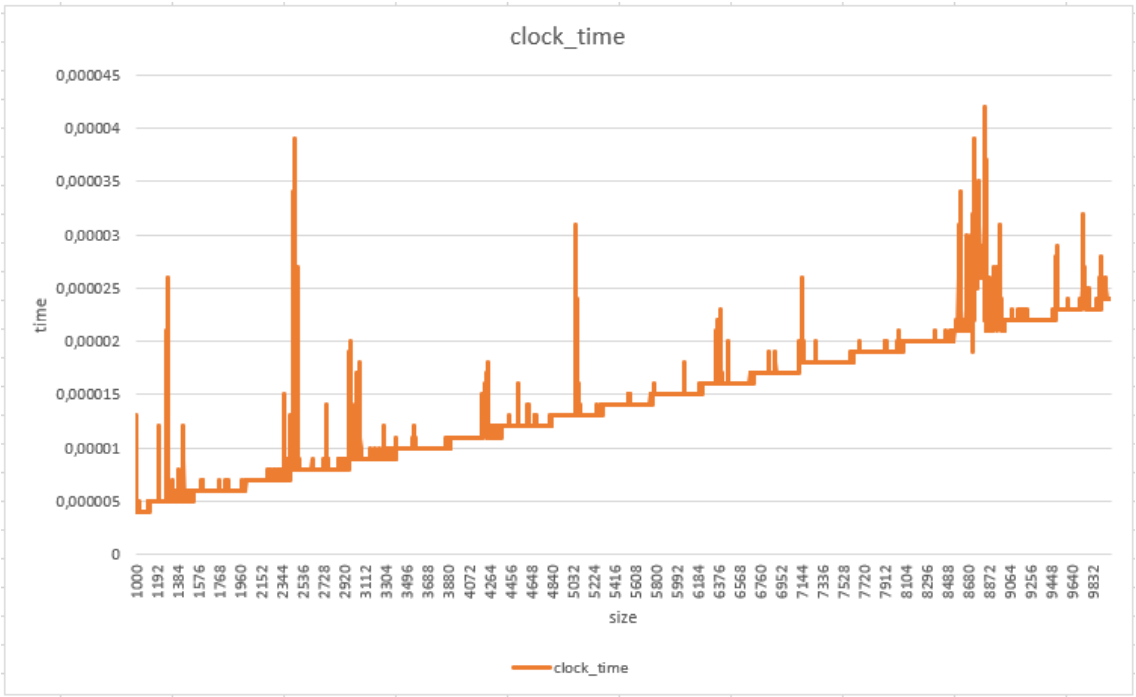


Plot comparing the average clock time for the linear and binary search approaches, comments to the plot.

Binary Search:

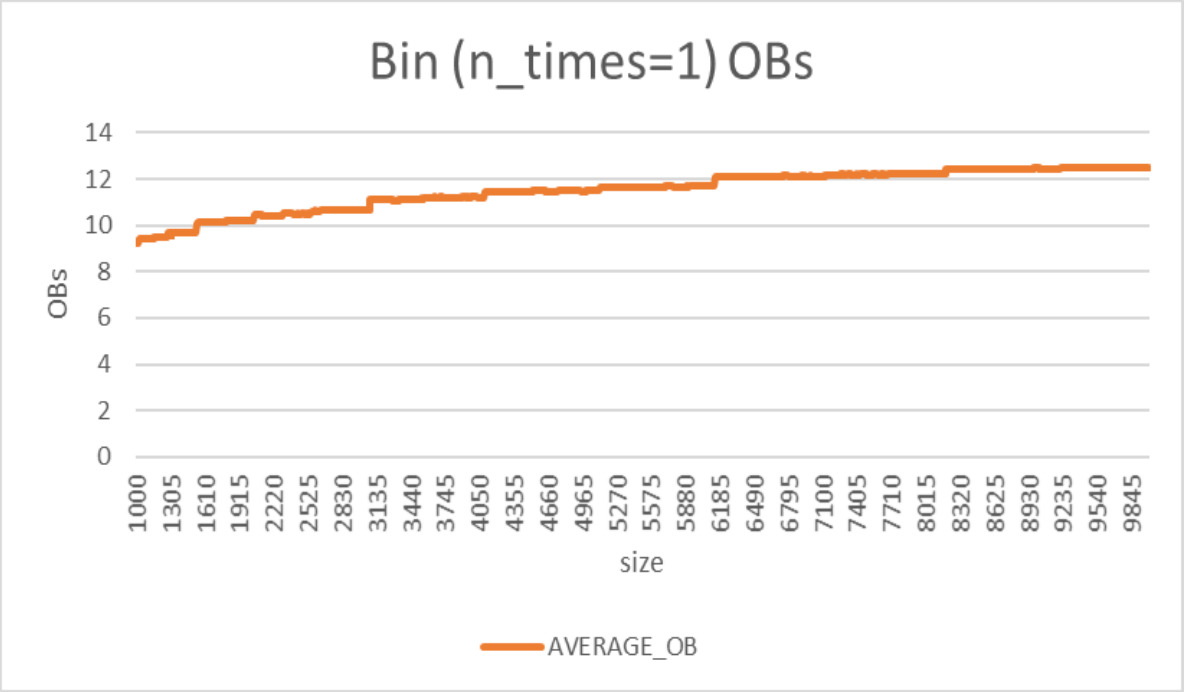


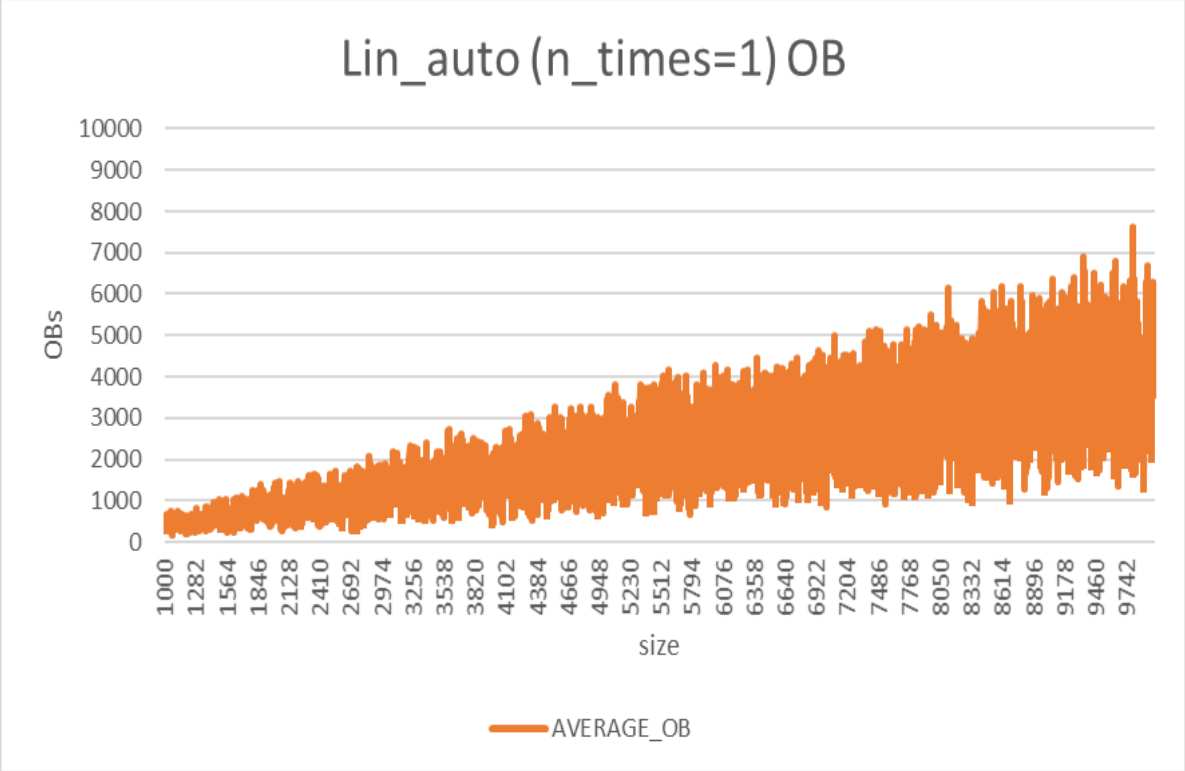
Linear Search:



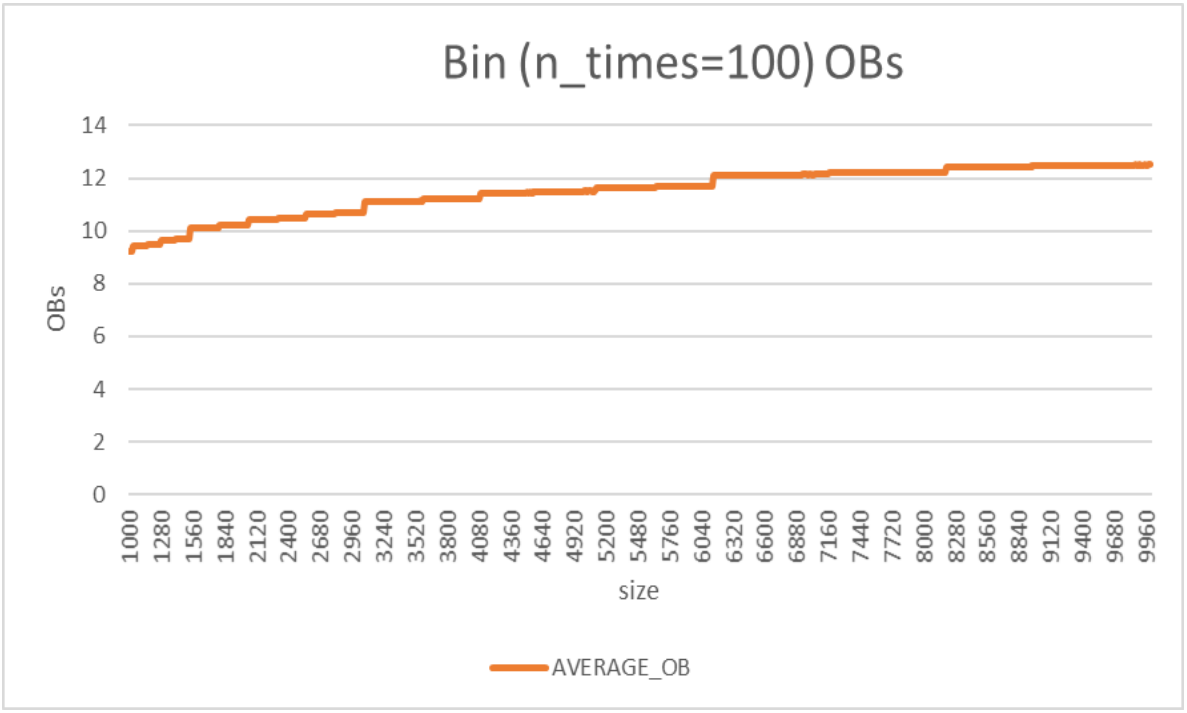
Plot comparing the average number of BOs of the binary and auto-organized linear search (for n_times=1, 100 y 10000), comments to the plot.

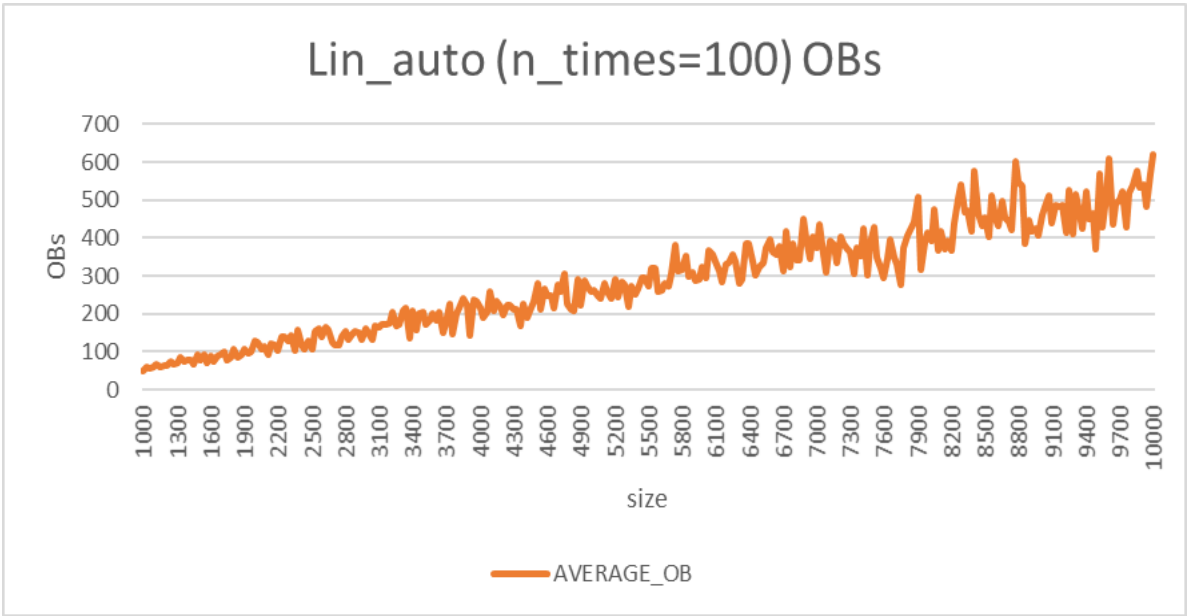
n_times=1



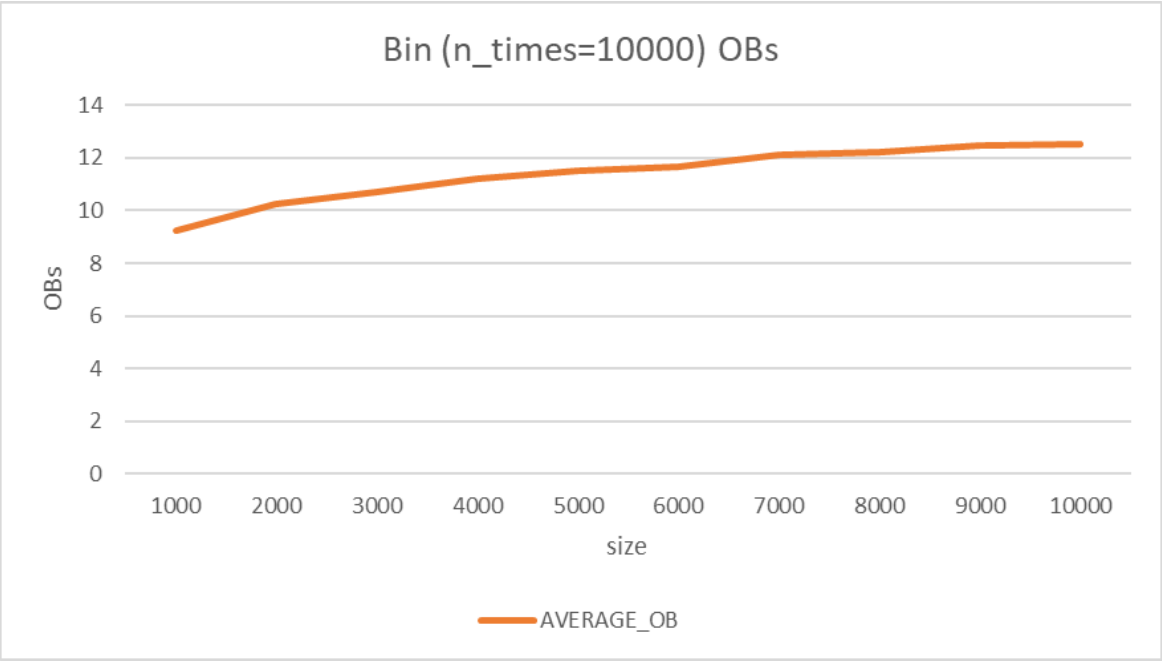


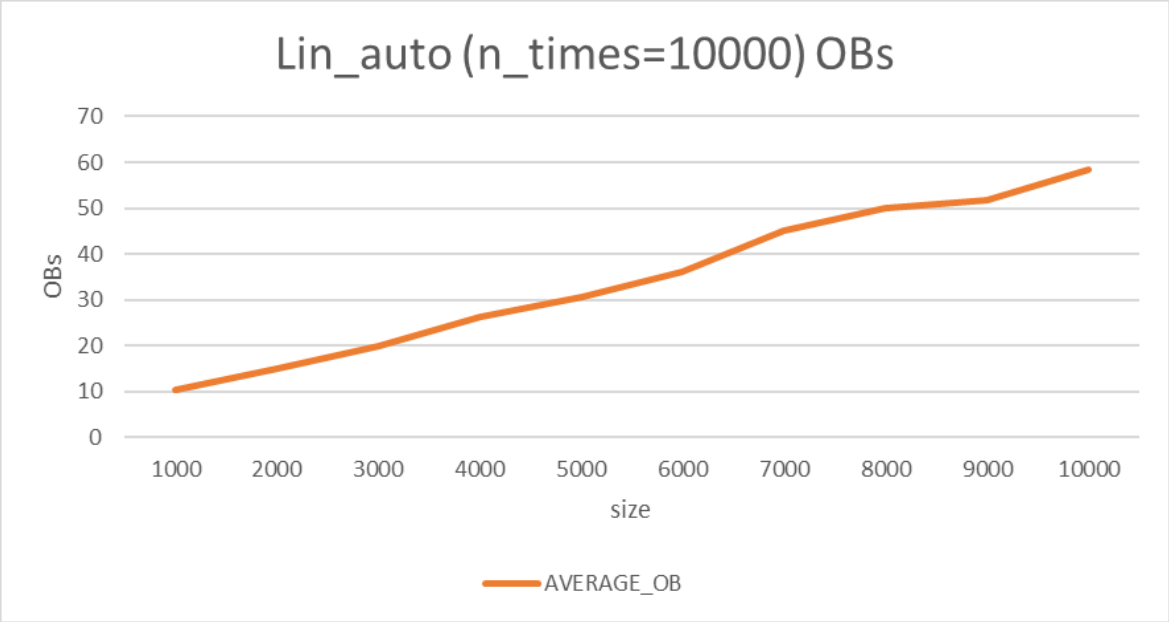
n_times=100


Bin (n_times=100) OBs
OBs
14
12
10
8
6
4
2
0
1000
1280
1560
1840
2120
2400
2680
2960
3240
3520
3800
4080
4360
4640
4920
5200
5480
5760
6040
6320
6600
6880
7160
7440
7720
8000
8280
8560
8840
9120
9400
9680
9960
size
AVERAGE_OB


Lin_auto (n_times=100) OBs
OBs
700
600
500
400
300
200
100
0
1000
1300
1600
1900
2200
2500
2800
3100
3400
3700
4000
4300
4600
4900
5200
5500
5800
6100
6400
6700
7000
7300
7600
7900
8200
8500
8800
9100
9400
9700
10000
size
AVERAGE_OB

n_times=10000

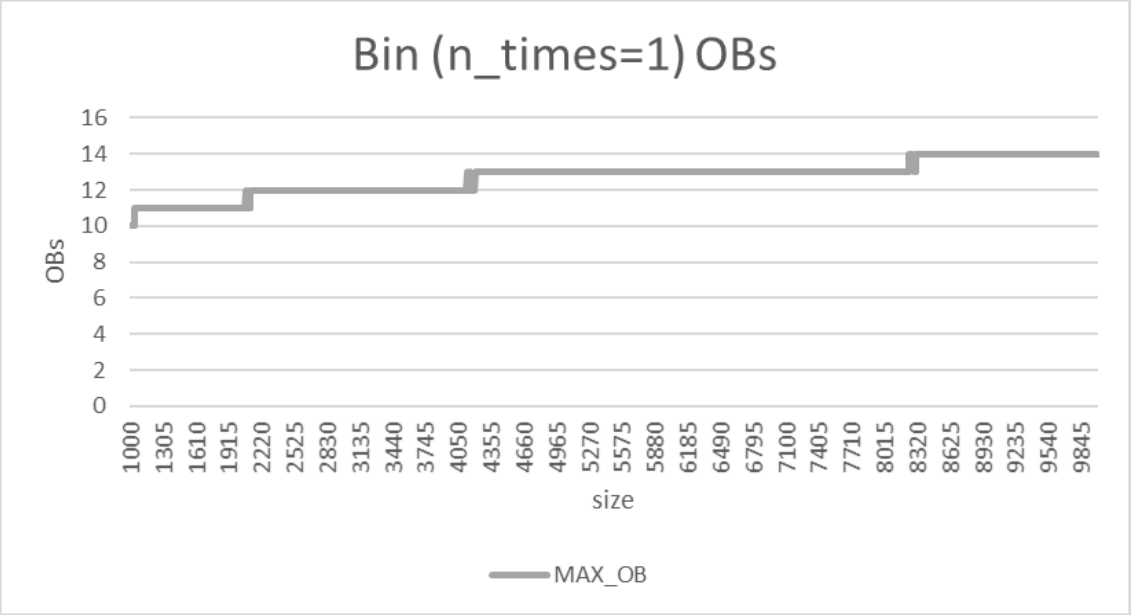


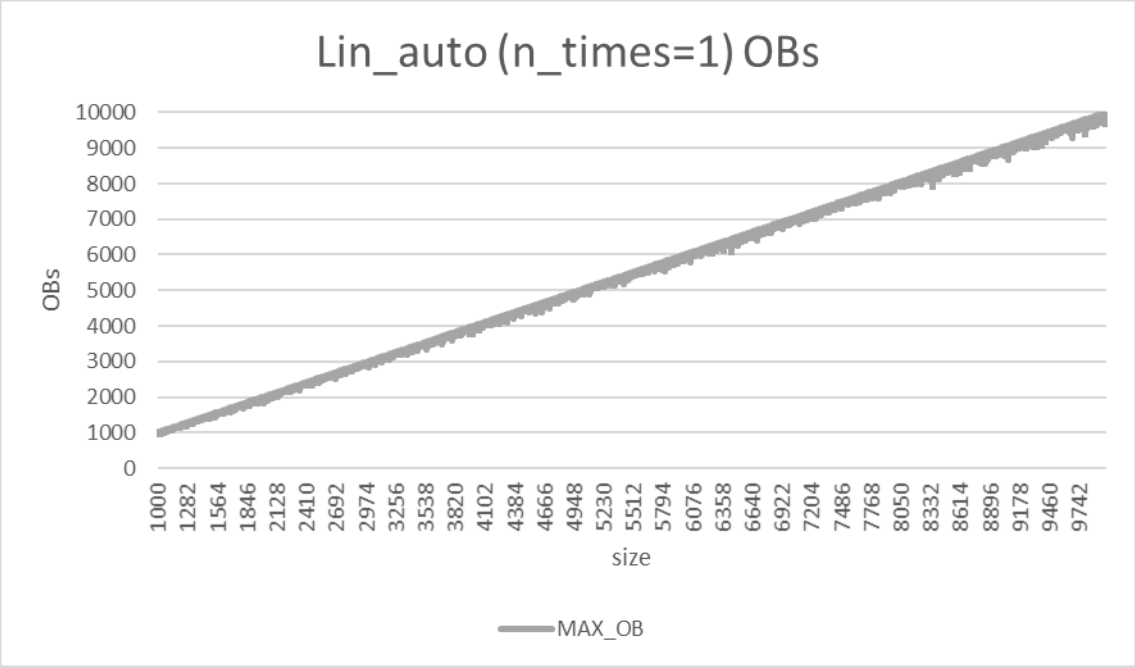


the average BOs for bin_search is a logarithmic function, while for lin_auto_search is a linear one. However, depending on n_times, the lin_auto_search lowers its BOs, while bin_search only improve its precision, but without changing the BOs. This happens because everytime we look for a certain key, the lin_auto_search places it one position before, making it 1 less BO the next time it looks for it. If we do it 10000 times, it's in the first position for a lot of them, which doesn't improve the maximum BOs, but improves a lot the average.

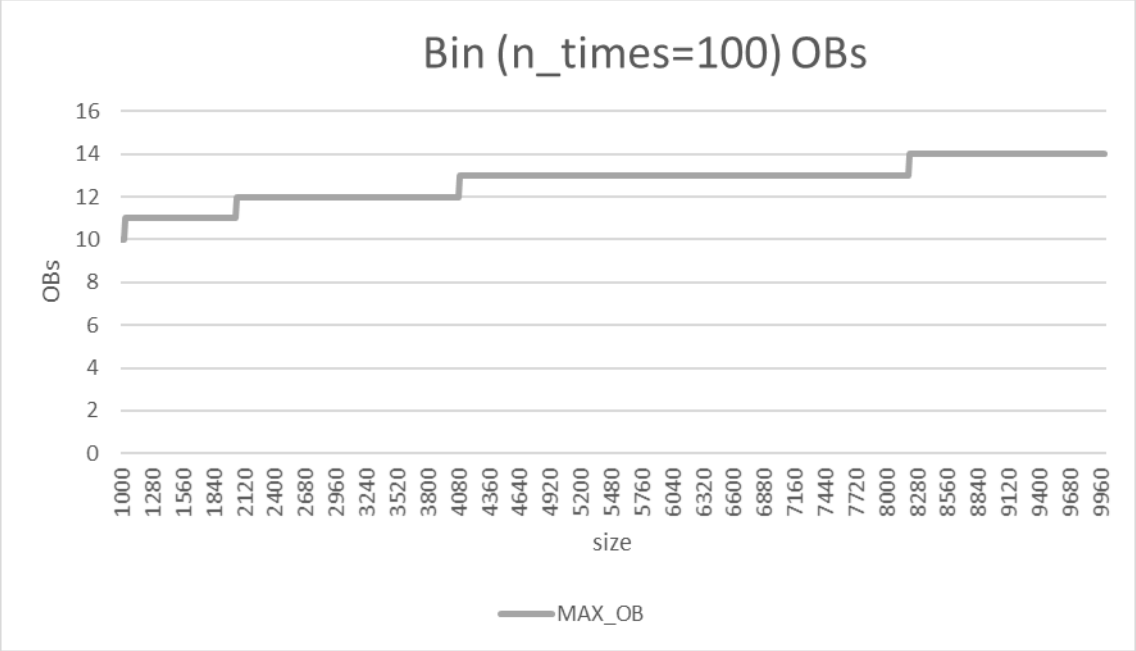
Plot comparing the maximum number of BOs of the binary and auto-organized linear search (for n_times=1, 100 y 10000), comments to the plot.

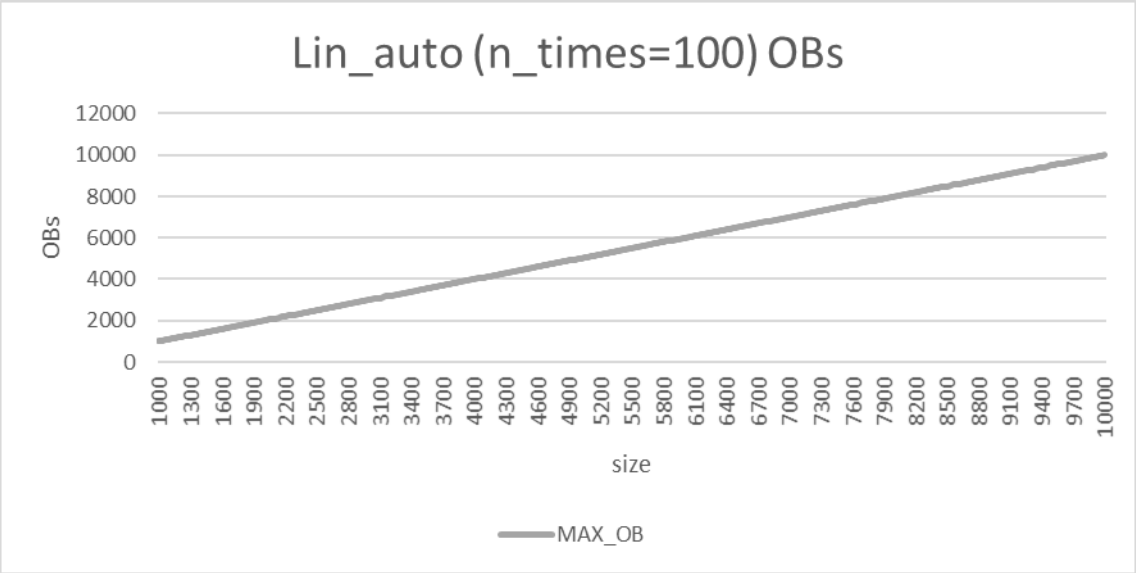
n_times=1



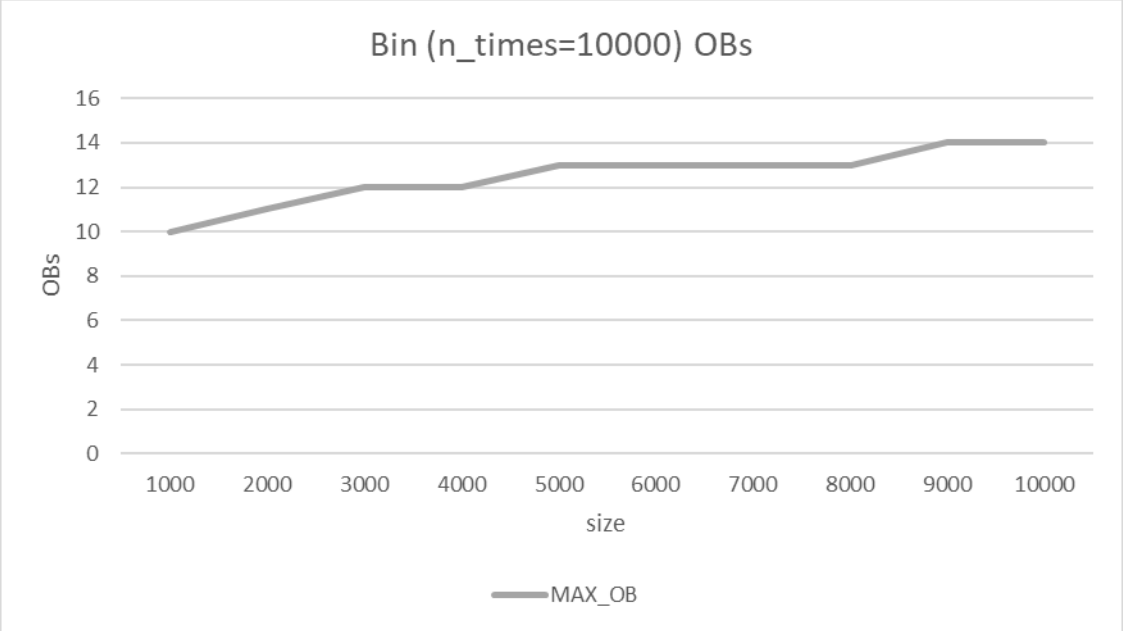


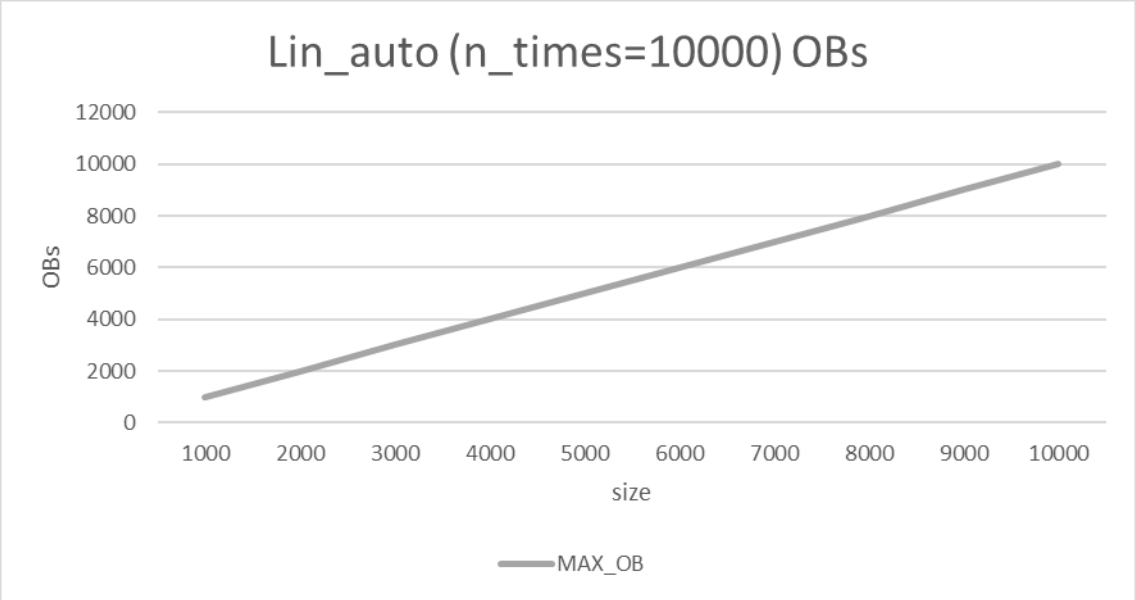
n_times=100


Bin (n_times=100) OBs
16
14
12
10
8
6
4
2
0
OBs
1000
1280
1560
1840
2120
2400
2680
2960
3240
3520
3800
4080
4360
4640
4920
5200
5480
5760
6040
6320
6600
6880
7160
7440
7720
8000
8280
8560
8840
9120
9400
9680
9960
size
MAX_OB


Lin_auto (n_times=100) OBs
12000
10000
8000
6000
4000
2000
0
OBs
1000
1300
1600
1900
2200
2500
2800
3100
3400
3700
4000
4300
4600
4900
5200
5500
5800
6100
6400
6700
7000
7300
7600
7900
8200
8500
8800
9100
9400
9700
10000
size
MAX_OB

n_times=10000

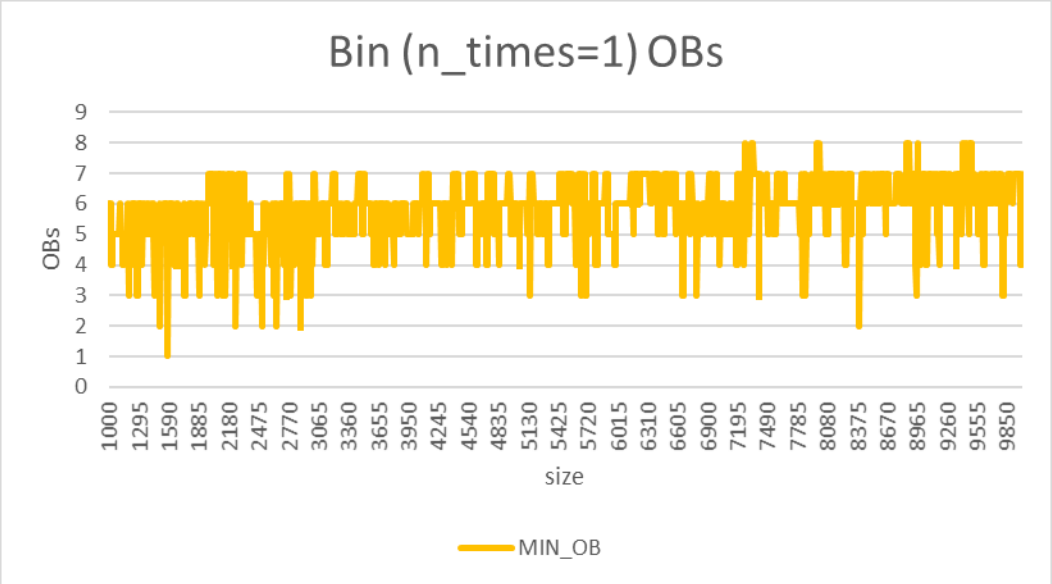


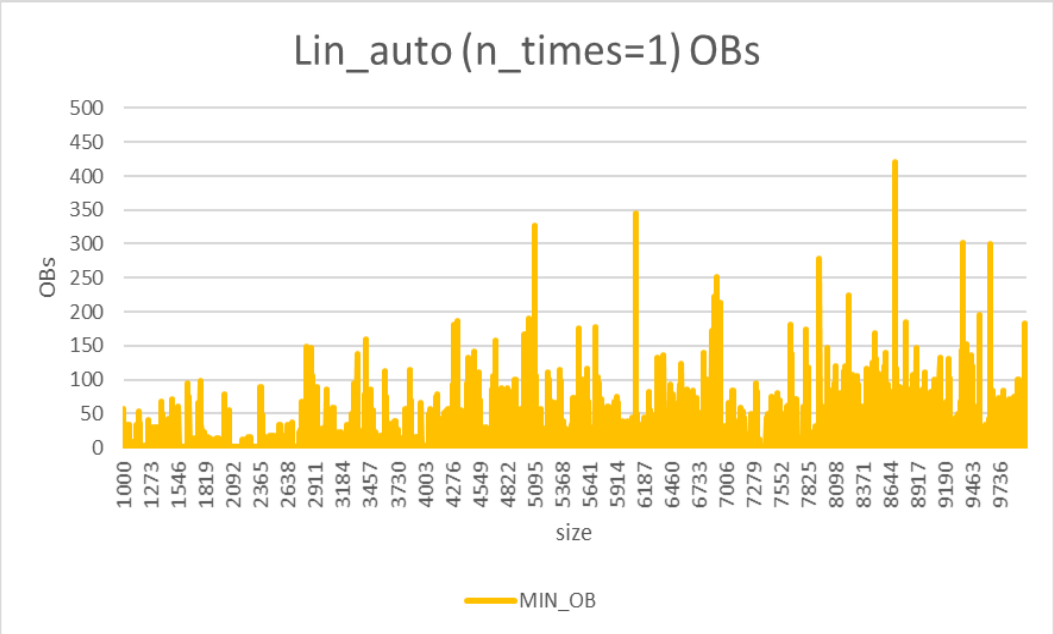


as we can see in these plots, lin_auto_search tends to have n as maximum, while bin_search tends to have log(n).

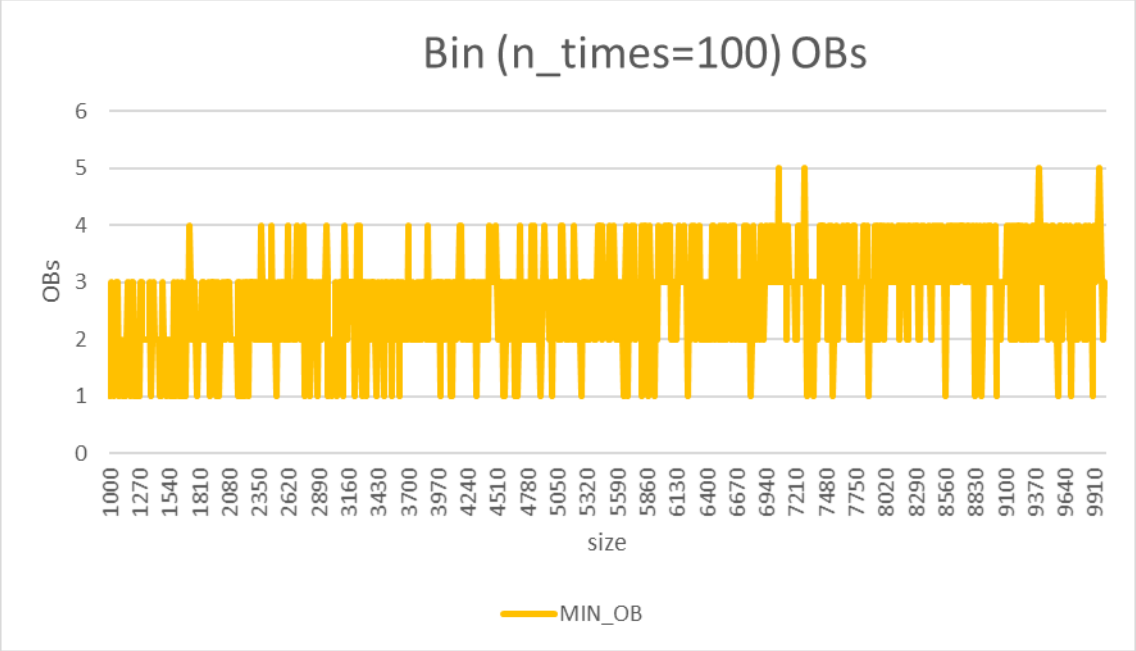
Plot comparing the minimum number of BOs of the binary and auto-organized linear search (for n_times=1, 100 y 10000), comments to the plot.

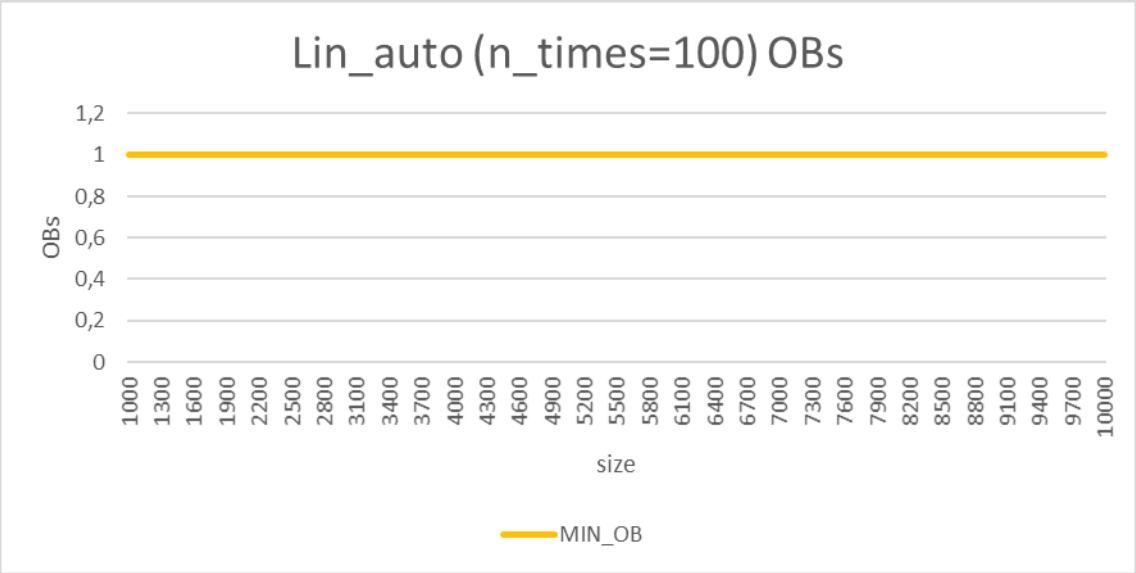
n_times=1



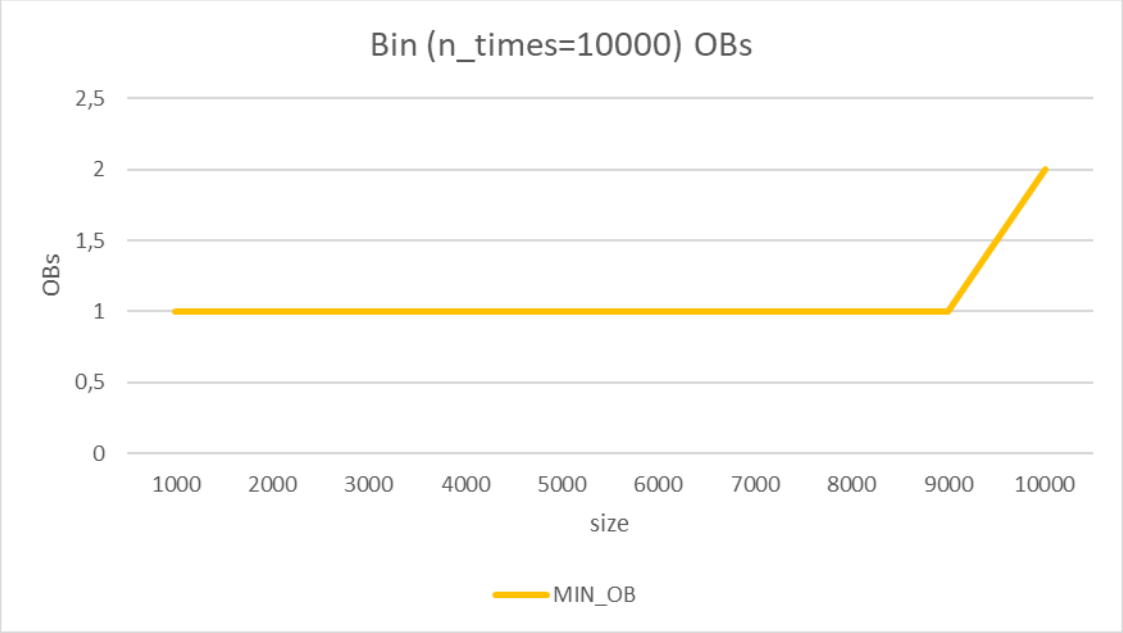


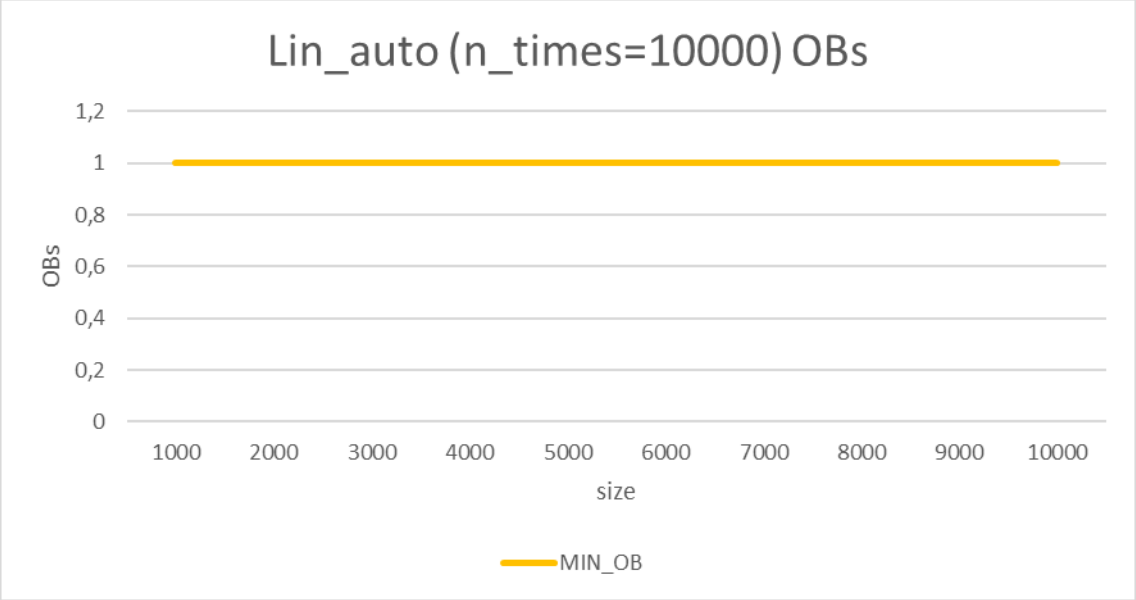
n_times=100


Bin (n_times=100) OBs
OBs
size
MIN_OB


Lin_auto (n_times=100) OBs
OBs
size
MIN_OB

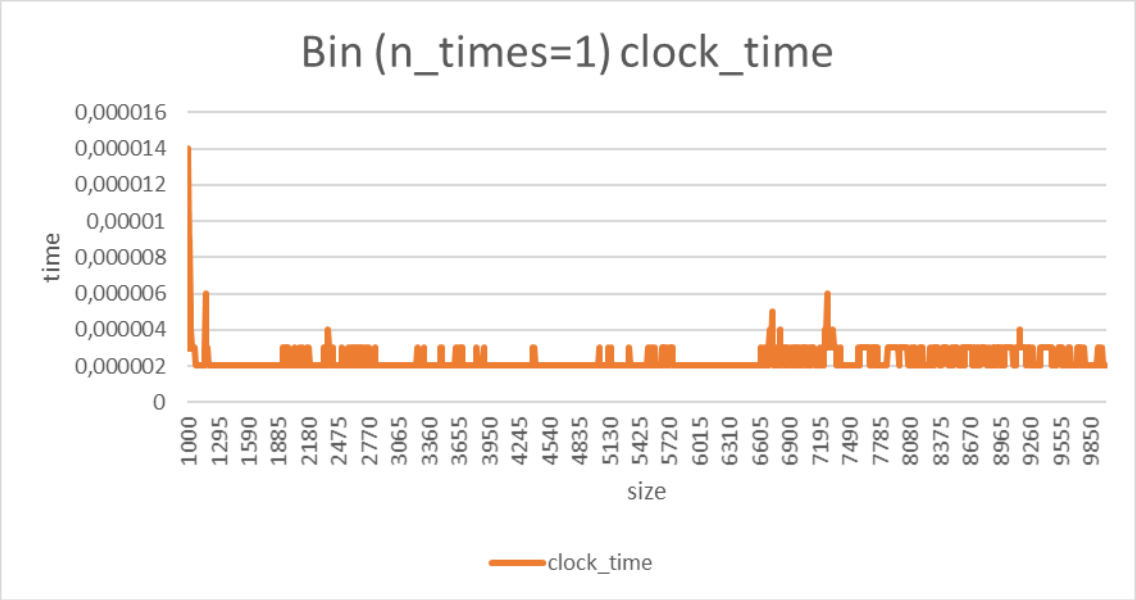
n_times=10000

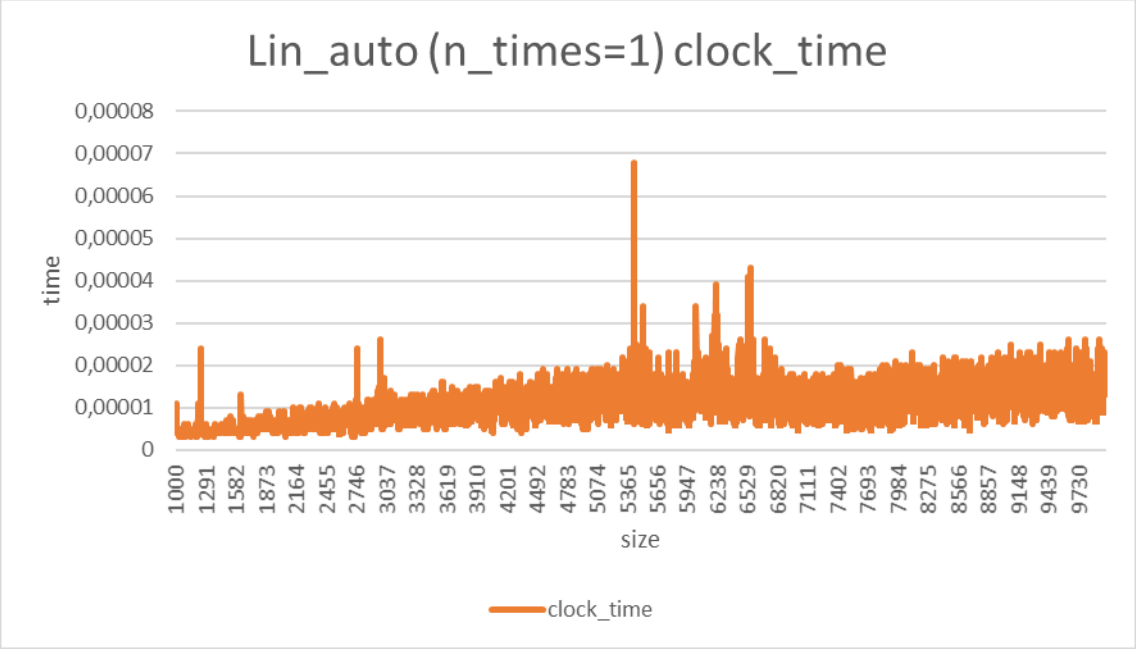




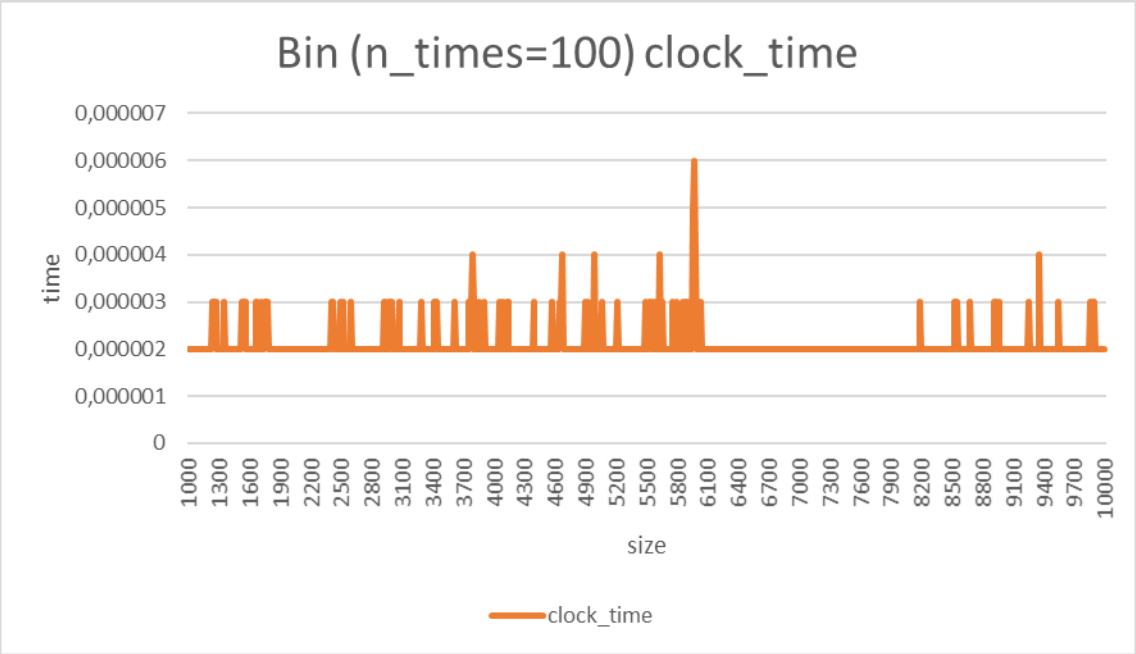
Plot comparing the average clock time for the binary and auto-organized linear search (for n_times=1, 100 y 10000), comments to the plot.

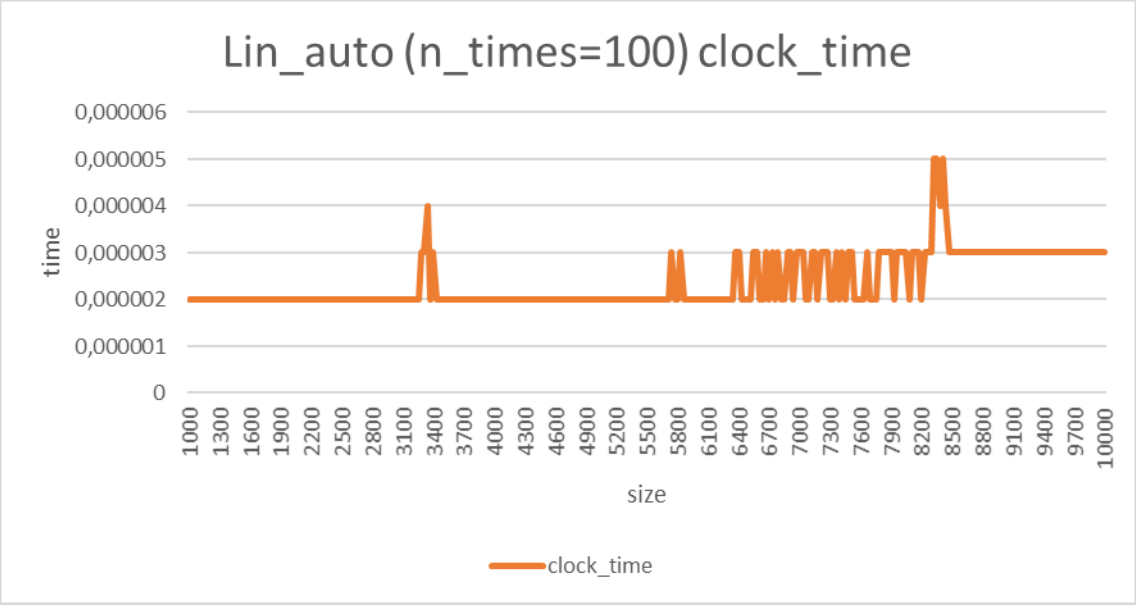
n_times=1



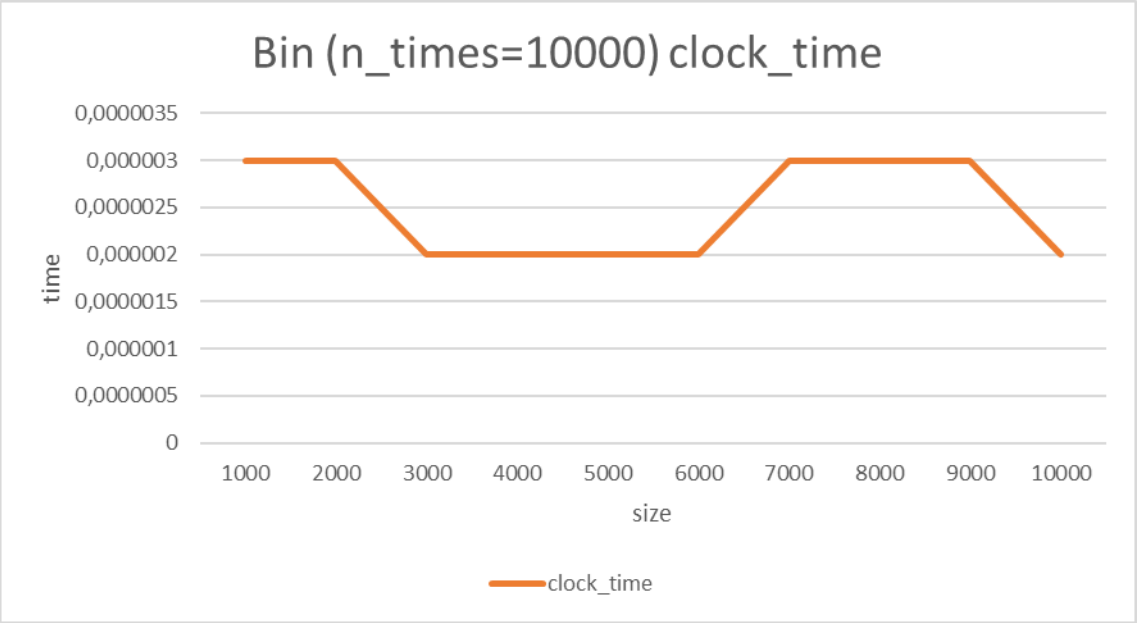


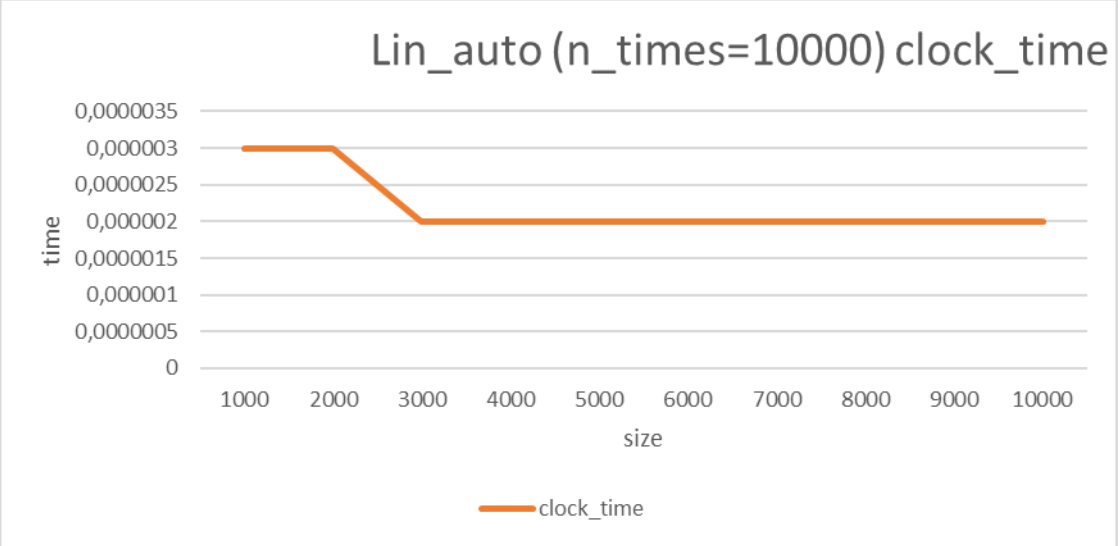
n_times=100


Bin (n_times=100) clock_time
0,000007
0,000006
0,000005
0,000004
0,000003
0,000002
0,000001
0
time
1000 1300 1600 1900 2200 2500 2800 3100 3400 3700 4000 4300 4600 4900 5200 5500 5800 6100 6400 6700 7000 7300 7600 7900 8200 8500 8800 9100 9400 9700 10000
size
clock_time


Lin_auto (n_times=100) clock_time
0,000006
0,000005
0,000004
0,000003
0,000002
0,000001
0
time
1000 1300 1600 1900 2200 2500 2800 3100 3400 3700 4000 4300 4600 4900 5200 5500 5800 6100 6400 6700 7000 7300 7600 7900 8200 8500 8800 9100 9400 9700 10000
size
clock_time

n_times=10000





5. Response to the theoretical questions.

Our answers to the theoretical questions.

5.1. Which is the basic operation of lin search, bin search and lin auto search?

In bin_search is this if:

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

In lin_search is this if:

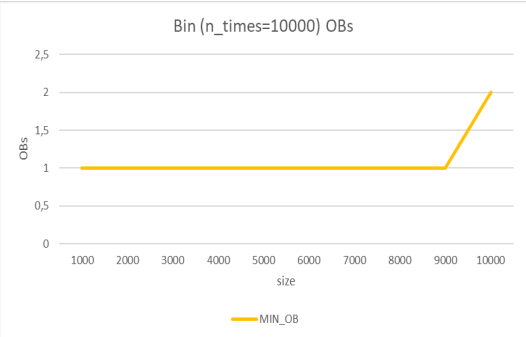
```
if(table[i]==key){
      flag=1;
```

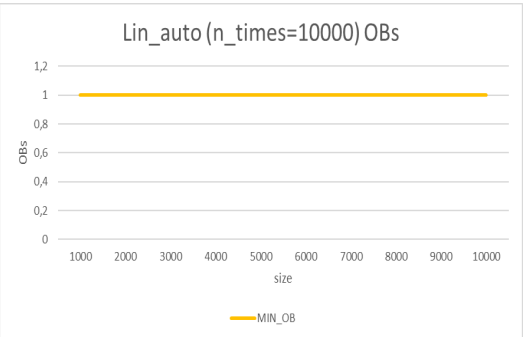
In lin_auto_search is this if:

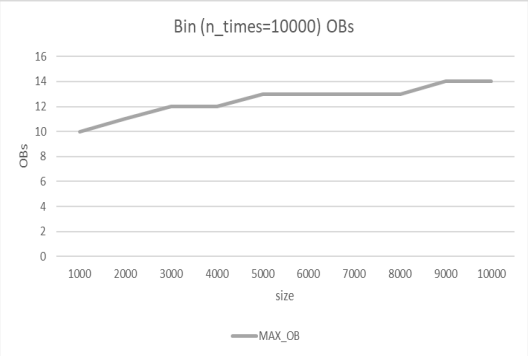
```
if(table[i]==key){
      if(i==0){*ppos=++i; return BOs;}
      flag=1;
    }
```

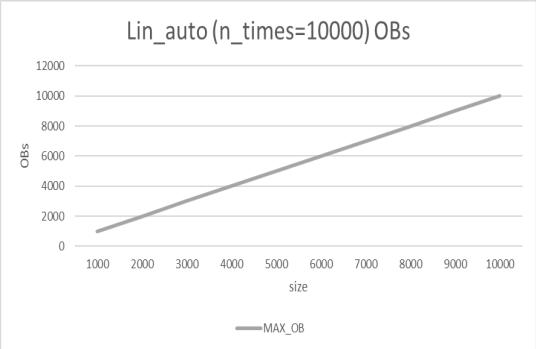
In all of them, it is the place of the function where it compares the value of the position in the table with the key.

5.2. Give the execution times, in terms of the input size n for the worst WSS(n) and best BSS(n) cases of bin search and lin search. Use the asymptotic notation (O, Θ, o, Ω,etc) as long as you can.









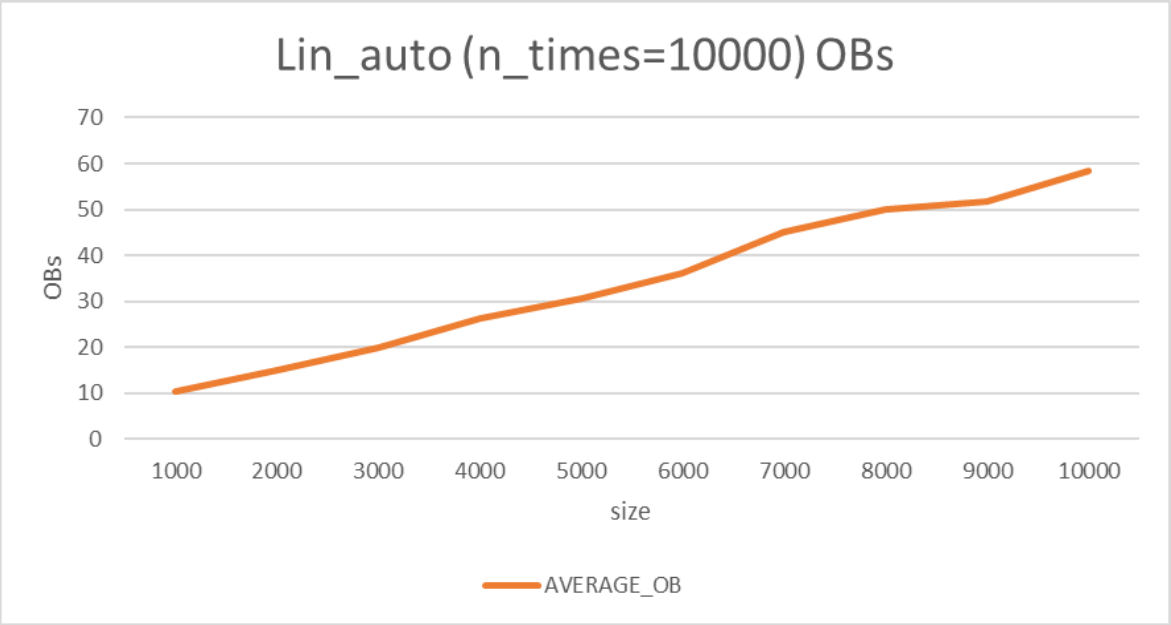
minBinET = kn (0 < k << 1)
minLinAutoET = 1
maxBinET = log(n)
maxLinAutoET = n

minBinET = O(minLinAutoET)
maxBinET = o(maxLinAutoET)

5.3. When lin auto search and the given not-uniform key distribution are used, how does it vary the position of the elements in the list of keys as long as the number of searches increases?

It kind of orders the table, because the lower numbers tend to go to the beginning of the table, since they are the most generated keys. The more number of searches, the more ordered it will be.

5.4. Which is the average execution time of lin auto search as a function of the number of elements in the dictionary n for the given not uniform key distribution? Consider that a large number of searches have been conducted and the list is in a quite stable state.



lin_auto_search = k·n (k ≃ 2/3)

5.5. Justify as formally as you can the correction (in other words, why it searches well) of the bin search algorithm.

Having L(n) as the left child of n and R(n) as the right one, we know that:

If the tree is well formed ∀n: n is part of the tree, L(n)<n ∧ R(n)>n

key = n

key<n ⇒ n := L(N)

key>n ⇒ n := R(N)

So, knowing we have all the numbers from 1 to N and the key is also between 1 and N, for any case it will end up in the key = n statement.

6. Conclusions.

With this practice we learned how to implement search algorithms for searching in a dictionary and also how to measure their efficiency. This was an interesting task for us, because we learnt which algorithms are better for each case.

Now we know that if the keys are usually the same, it's better to use a linear auto-organized search algorithm, but in other cases the binary search algorithm ends up being the most efficient one.