Analysis of Algorithms 2019/2020

Practice 3

Álvaro Castillo & Pablo Ernesto Soëtard, Group 1291

Code	Plots	Documentatio n	Total

1. Introduction.

In this practice, the development and analysis of search algorithms over dictionaries is studied. Specifically, the average time to search elements over a dictionary which uses as data type a table will be experimentally obtained and compared to the theoretical data.

2. Objectives

The first exercise is related to the programming of searching functions, while the second one with the extraction of data.

2.1 Section 1

We are going to create this functions:

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

void free_dictionary(PDICT pdict);

int insert_dictionary(PDICT pdict,int key);

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

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

int bin_search(int *table,int F,int L,int key,int *ppos);

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

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

The first five functions manage dictionaries and the last three are different searching methods. `init_dictionary` will create a dictionary, allocating memory for it, while `free_dictionary` will delete it, freeing all the resources. `massive_insertion_dictionary` will call `insert_dictionary` to insert recursively all the keys. `insert_dictionary` will insert into our pdict the kay that it recieves.

`bin_search`, `lin_search` and `lin_auto_search` are going to search a key into a dictionary with the method implemented (Binary Search, Linear Search and Linear Auto Search respectively).

2.2 Section 2

In this section we will program this functions:

```
short average_search_time(pfunc_busqueda method, pfunc_key_generator
generator, char order, int N, int n_times, PTIME_AA ptime);

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);
```

As we have done in the first practice, this two functions will generate the data that will be plotted. The first function will create a dictionary and a permutation of keys. Then it will insert the keys into the dictionary and it will call to the method function to search all the keys. The second one will allocate ptime structures and call to `average_search_time` several times to fill these structures. Then it will call to `save_time_table` to print the data in the structures. Finally it will free the ptime structures.

3. Tools and methodology

Our environment to program is Ubuntu. The tools that we have used are Atom and GitKraken to code, as well as github to share our code and Valgrind to check the memory leaks. We have also used gnuplot to plot the functions and gcc to compile all the files.

3.1 Section 1

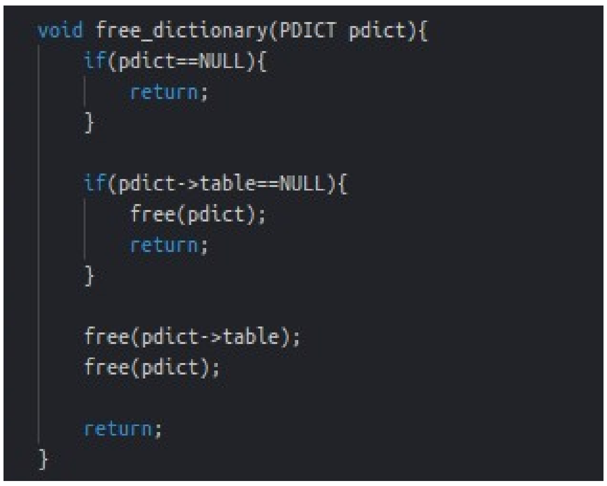
Linear search was easy to program as well as linear auto search because both were really intuitive. Binary search was more difficult, as we had implemented it as a recursive function. We had problems counting the number of basic operations and with the base case for the returning comparison. Functions for initialazing, freeing and searching on the dictionary, were also very easy to implement. We had to think more for the data insertion on the dictionary, but it was not a big problem.

3.2 Section 2

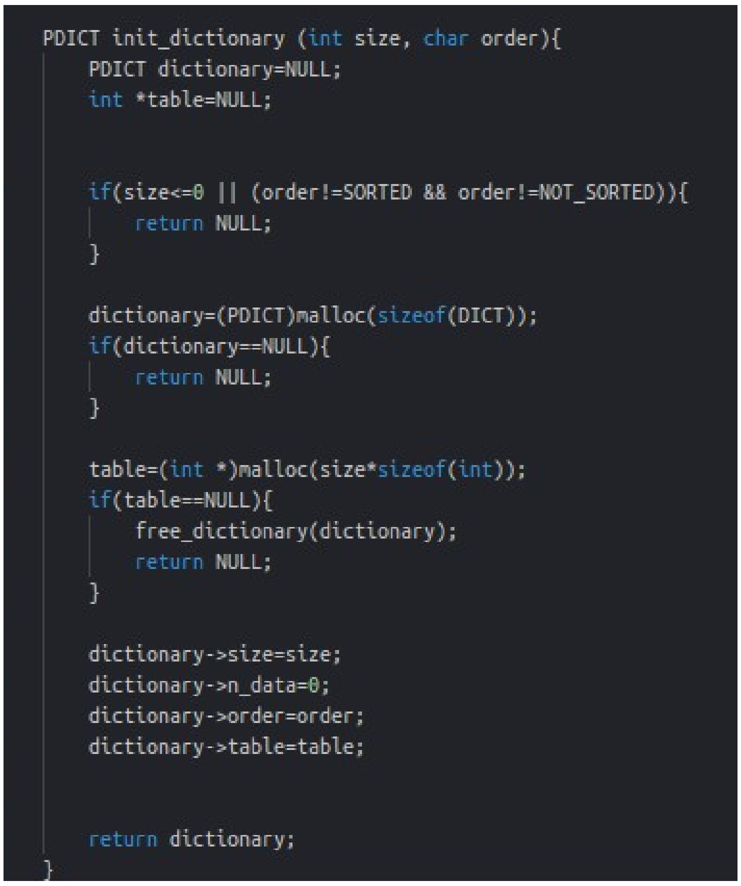
The function for generating all the ptime structures was practically the same as the one that we had for the sorting algorithms, so it was easy to create. Average_search_time was more challenging to code because it had to call a lot of functions and check several values. We had a problem with the loop that calls to the method and then calculating the avearge time an number of BO, as we had to execute it N*n_times and not N as in the other practice. We had also to do a little change into the function save_time_table in order to have more decimals on the average time.

4. Source code

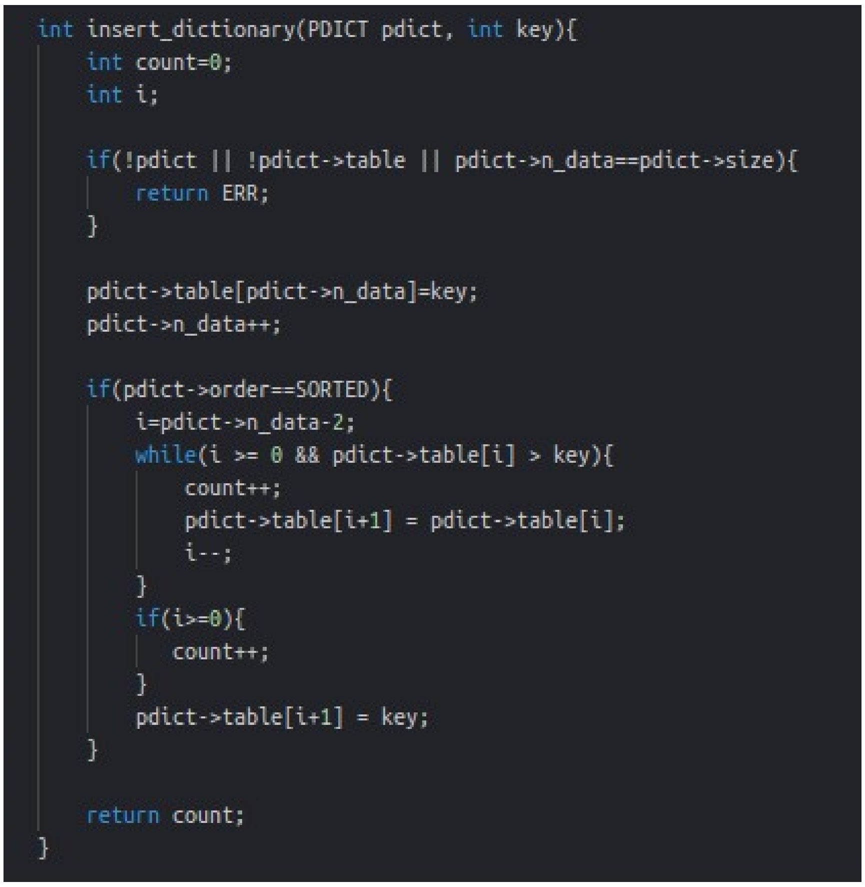
4.1 Section 1



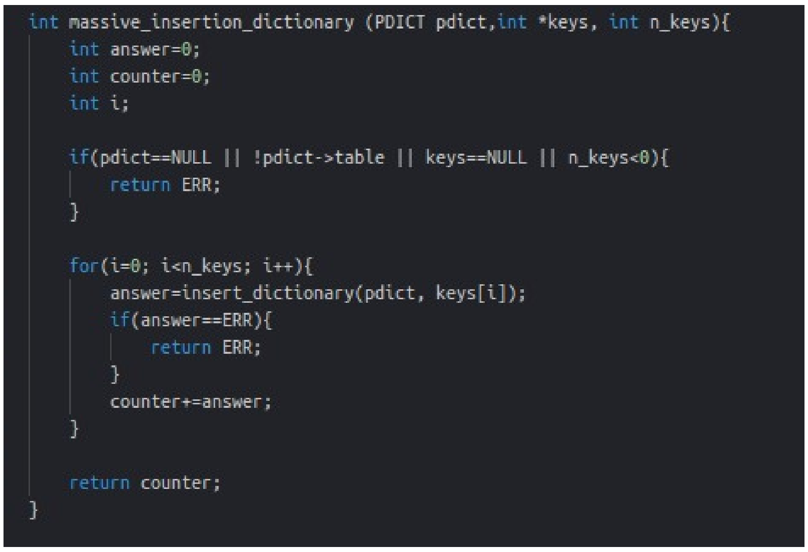
```
void free_dictionary(PDICT pdict){
    if(pdict==NULL){
        return;
    }

    if(pdict->table==NULL){
        free(pdict);
        return;
    }

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

    return;
}
```



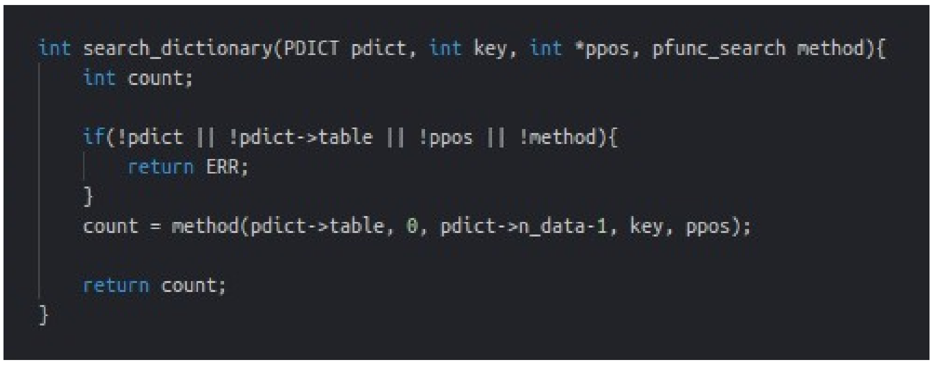
```c
PDICT init_dictionary (int size, char order){
    PDICT dictionary=NULL;
    int *table=NULL;


    if(size<=0 || (order!=SORTED && order!=NOT_SORTED)){
        return NULL;
    }

    dictionary=(PDICT)malloc(sizeof(DICT));
    if(dictionary==NULL){
        return NULL;
    }

    table=(int *)malloc(size*sizeof(int));
    if(table==NULL){
        free_dictionary(dictionary);
        return NULL;
    }

    dictionary->size=size;
    dictionary->n_data=0;
    dictionary->order=order;
    dictionary->table=table;


    return dictionary;
}
```



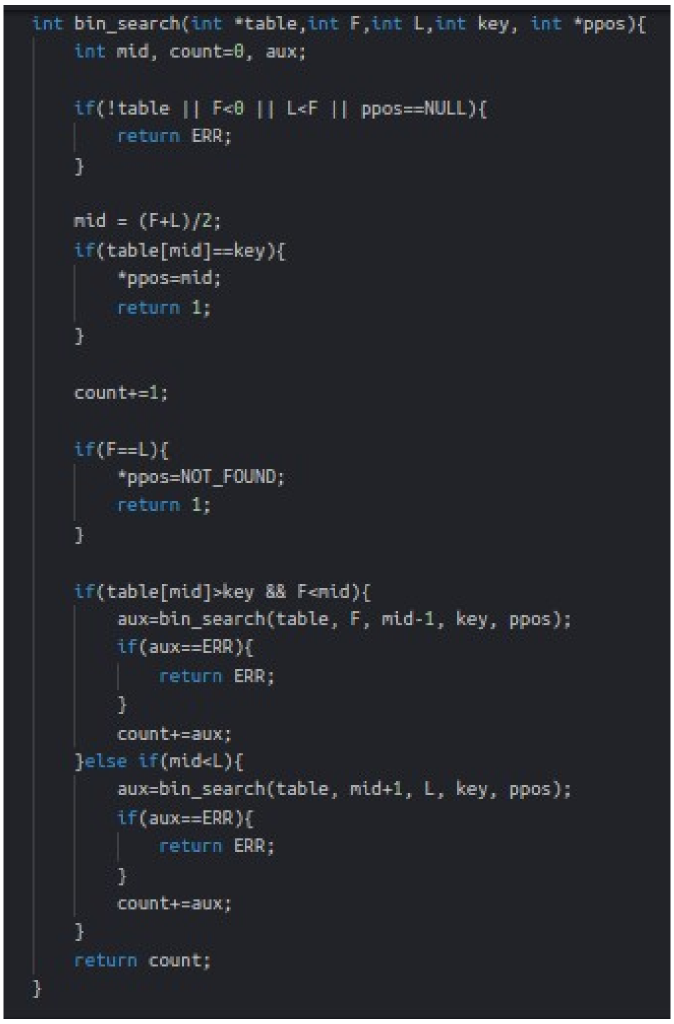
```c
int insert_dictionary(PDICT pdict, int key){
    int count=0;
    int i;

    if(!pdict || !pdict->table || pdict->n_data==pdict->size){
        return ERR;
    }

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

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

    return count;
}
```



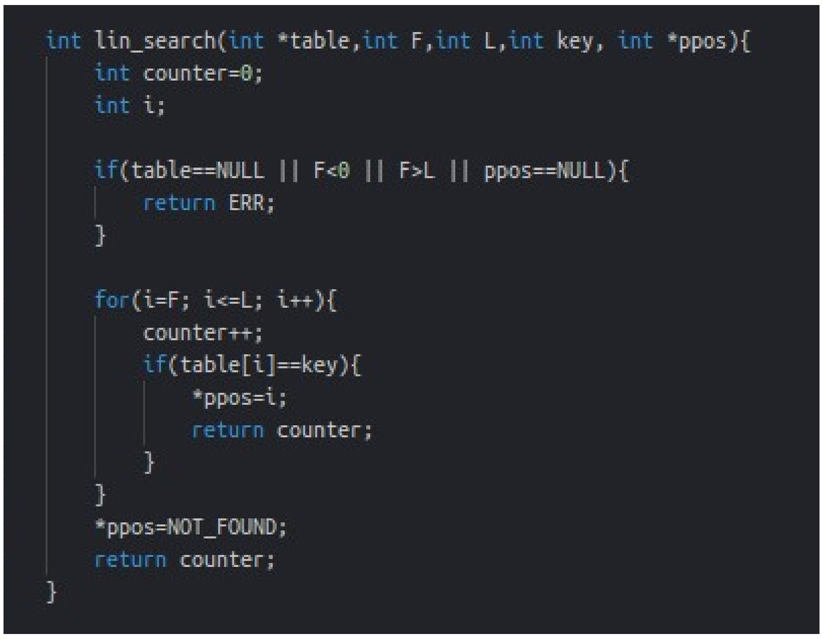
```c
int massive_insertion_dictionary (PDICT pdict,int *keys, int n_keys){
    int answer=0;
    int counter=0;
    int i;

    if(pdict==NULL || !pdict->table || keys==NULL || n_keys<0){
        return ERR;
    }

    for(i=0; i<n_keys; i++){
        answer=insert_dictionary(pdict, keys[i]);
        if(answer==ERR){
            return ERR;
        }
        counter+=answer;
    }

    return counter;
}
```



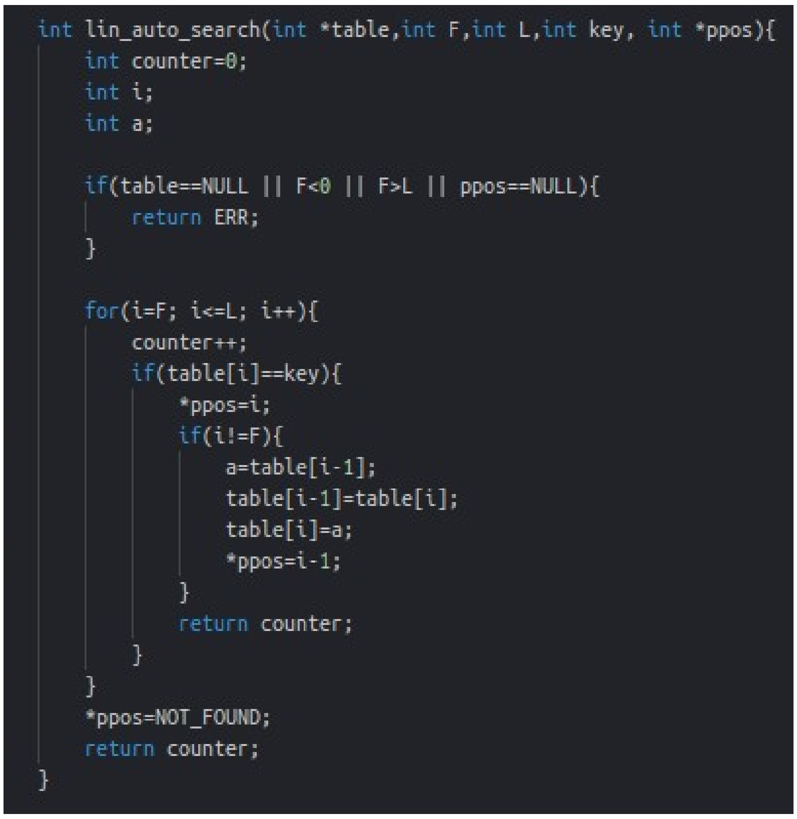
```c
int search_dictionary(PDICT pdict, int key, int *ppos, pfunc_search method){
    int count;

    if(!pdict || !pdict->table || !ppos || !method){
        return ERR;
    }
    count = method(pdict->table, 0, pdict->n_data-1, key, ppos);

    return count;
}
```



```c
int bin_search(int *table,int F,int L,int key, int *ppos){
    int mid, count=0, aux;

    if(!table || F<0 || L<F || ppos==NULL){
        return ERR;
    }

    mid = (F+L)/2;
    if(table[mid]==key){
        *ppos=mid;
        return 1;
    }

    count+=1;

    if(F==L){
        *ppos=NOT_FOUND;
        return 1;
    }

    if(table[mid]>key && F<mid){
        aux=bin_search(table, F, mid-1, key, ppos);
        if(aux==ERR){
            return ERR;
        }
        count+=aux;
    }else if(mid<L){
        aux=bin_search(table, mid+1, L, key, ppos);
        if(aux==ERR){
            return ERR;
        }
        count+=aux;
    }
    return count;
}
```

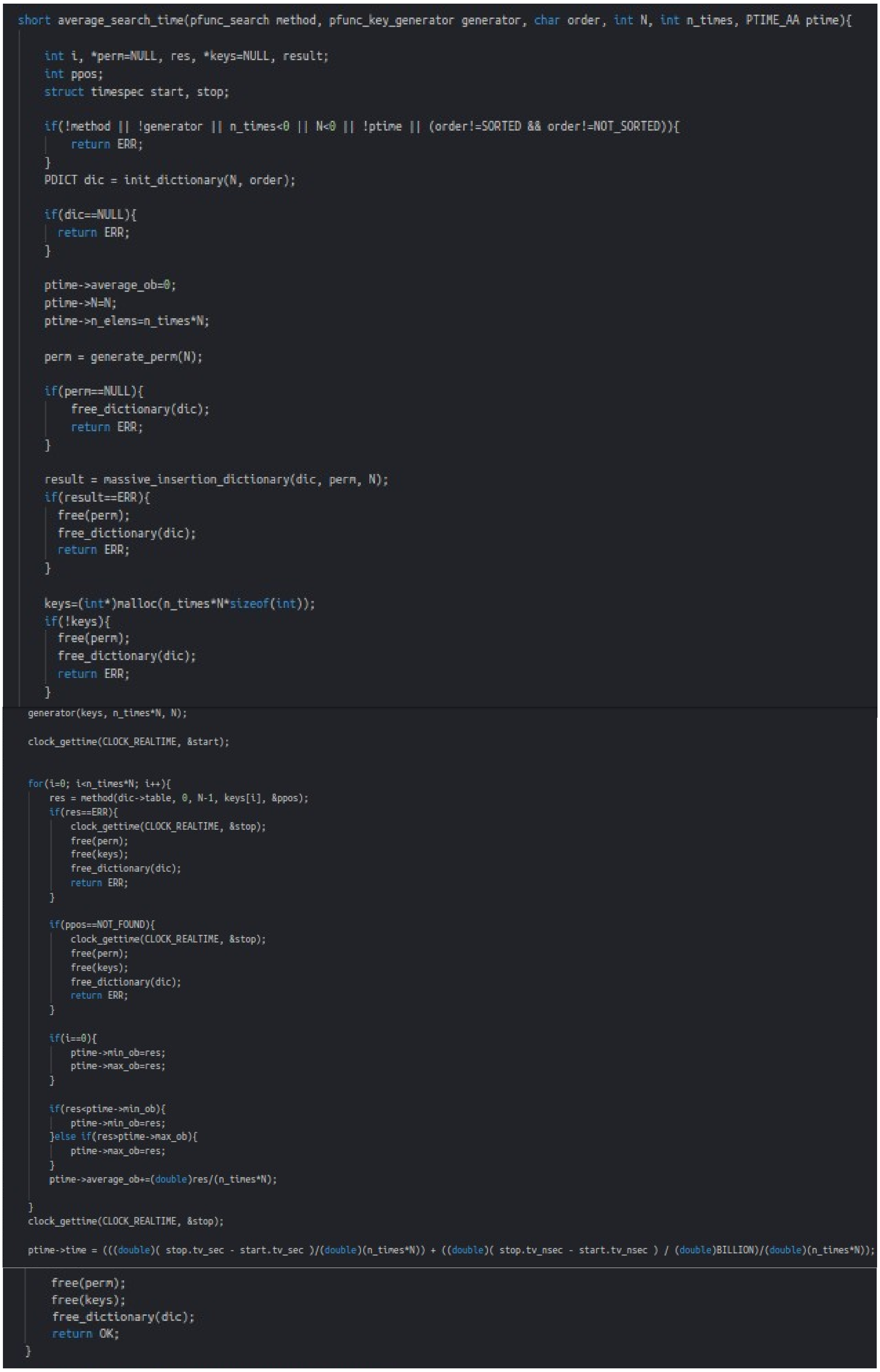


```c
int lin_search(int *table,int F,int L,int key, int *ppos){
    int counter=0;
    int i;

    if(table==NULL || F<0 || F>L || ppos==NULL){
        return ERR;
    }

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

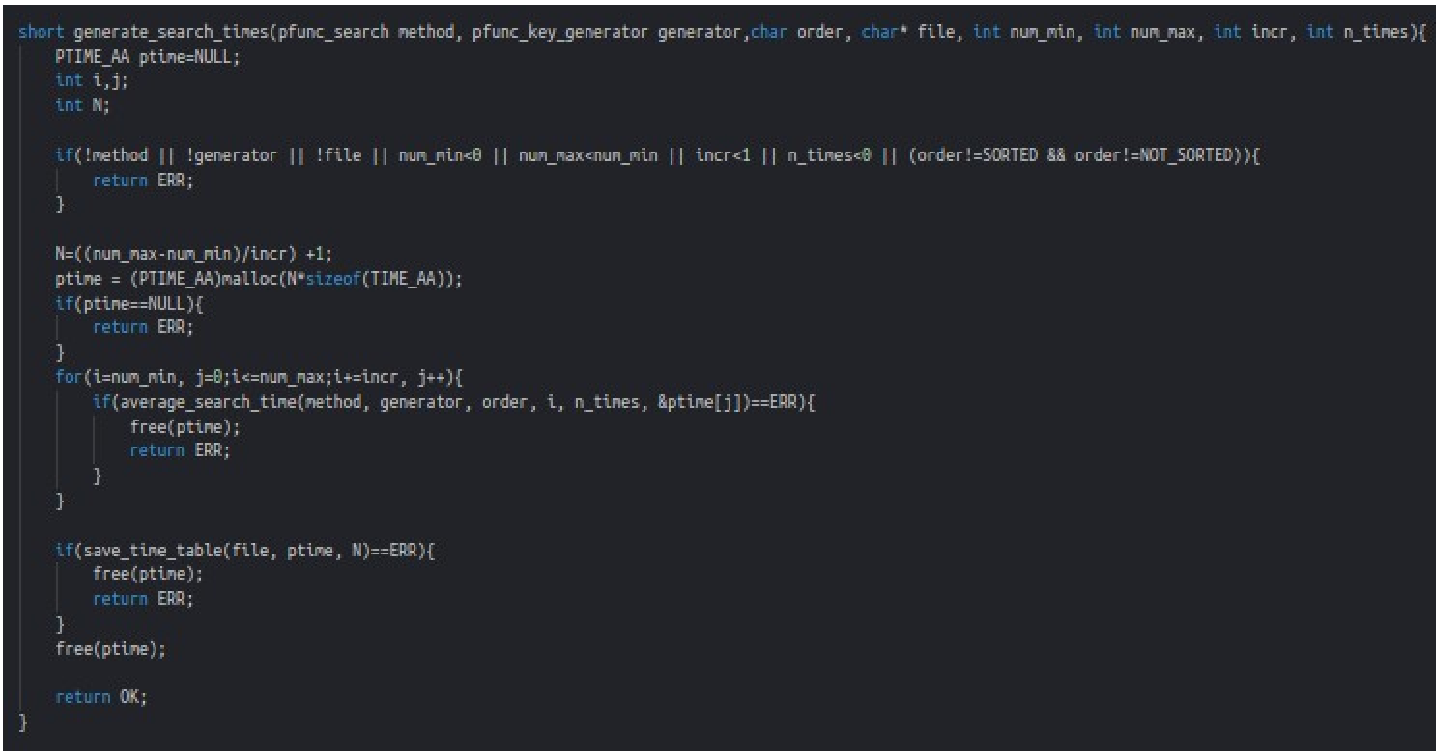


```c
int lin_auto_search(int *table,int F,int L,int key, int *ppos){
    int counter=0;
    int i;
    int a;

    if(table==NULL || F<0 || F>L || ppos==NULL){
        return ERR;
    }

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

4.2 Section 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, *perm=NULL, res, *keys=NULL, result;
    int ppos;
    struct timespec start, stop;

    if(!method || !generator || n_times<0 || N<0 || !ptime || (order!=SORTED && order!=NOT_SORTED)){
        return ERR;
    }
    PDICT dic = init_dictionary(N, order);

    if(dic==NULL){
      return ERR;
    }

    ptime->average_ob=0;
    ptime->N=N;
    ptime->n_elems=n_times*N;

    perm = generate_perm(N);

    if(perm==NULL){
        free_dictionary(dic);
        return ERR;
    }

    result = massive_insertion_dictionary(dic, perm, N);
    if(result==ERR){
      free(perm);
      free_dictionary(dic);
      return ERR;
    }

    keys=(int*)malloc(n_times*N*sizeof(int));
    if(!keys){
      free(perm);
      free_dictionary(dic);
      return ERR;
    }
  generator(keys, n_times*N, N);

  clock_gettime(CLOCK_REALTIME, &start);


  for(i=0; i<n_times*N; i++){
      res = method(dic->table, 0, N-1, keys[i], &ppos);
      if(res==ERR){
          clock_gettime(CLOCK_REALTIME, &stop);
          free(perm);
          free(keys);
          free_dictionary(dic);
          return ERR;
      }

      if(ppos==NOT_FOUND){
          clock_gettime(CLOCK_REALTIME, &stop);
          free(perm);
          free(keys);
          free_dictionary(dic);
          return ERR;
      }

      if(i==0){
          ptime->min_ob=res;
          ptime->max_ob=res;
      }

      if(res<ptime->min_ob){
          ptime->min_ob=res;
      }else if(res>ptime->max_ob){
          ptime->max_ob=res;
      }
      ptime->average_ob+=(double)res/(n_times*N);

  }
  clock_gettime(CLOCK_REALTIME, &stop);

  ptime->time = (((double)( stop.tv_sec - start.tv_sec )/(double)(n_times*N)) + ((double)( stop.tv_nsec - start.tv_nsec ) / (double)BILLION)/(double)(n_times*N));

    free(perm);
    free(keys);
    free_dictionary(dic);
    return OK;
}
```


```c
short generate_search_times(pfunc_search method, pfunc_key_generator generator,char order, char* file, int num_min, int num_max, int incr, int n_times){
    PTIME_AA ptime=NULL;
    int i,j;
    int N;

    if(!method || !generator || !file || num_min<0 || num_max<num_min || incr<1 || n_times<0 || (order!=SORTED && order!=NOT_SORTED)){
        return ERR;
    }

    N=((num_max-num_min)/incr) +1;
    ptime = (PTIME_AA)malloc(N*sizeof(TIME_AA));
    if(ptime==NULL){
        return ERR;
    }
    for(i=num_min, j=0;i<=num_max;i+=incr, j++){
        if(average_search_time(method, generator, order, i, n_times, &ptime[j])==ERR){
            free(ptime);
            return ERR;
        }
    }

    if(save_time_table(file, ptime, N)==ERR){
        free(ptime);
        return ERR;
    }
    free(ptime);

    return OK;
}
```

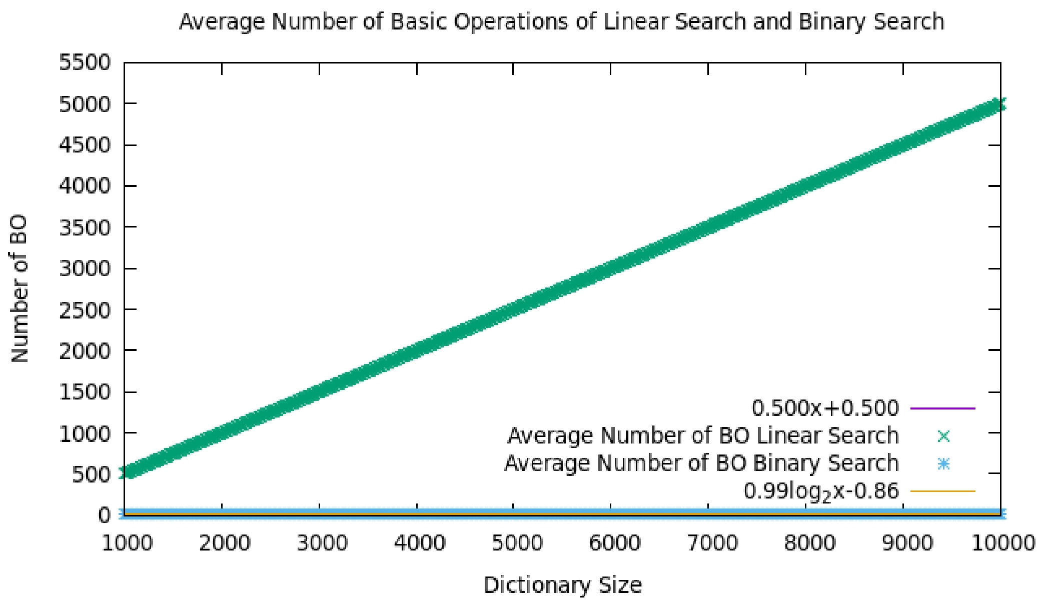
5. Results, plots

5.1 Section 1

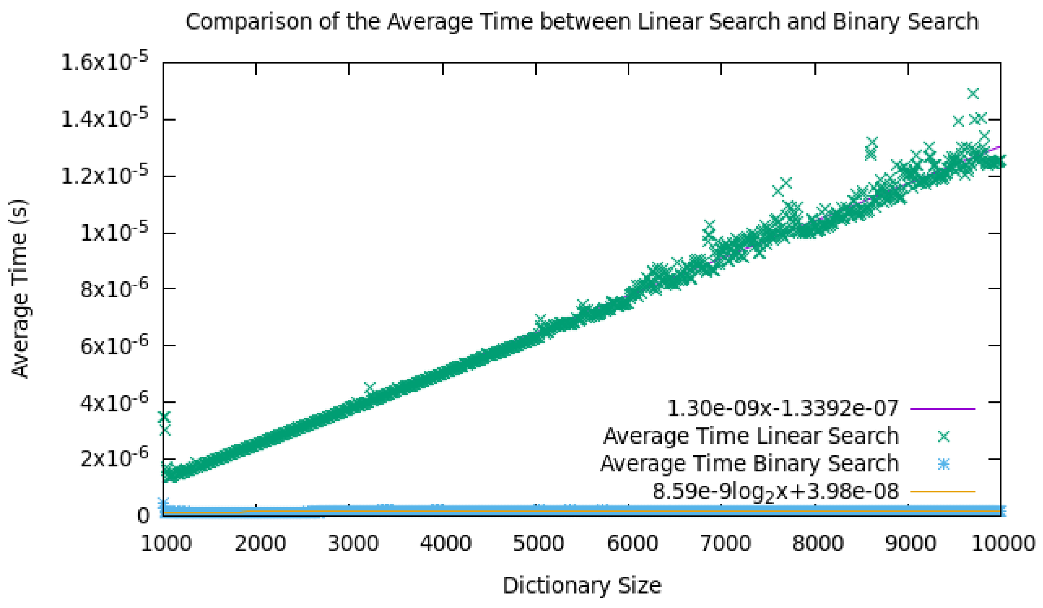
For this first section we have executed the c main file exercise1.c and we have checked that all functions worked correctly and that our three programmed searching methods gave the correct output.

5.2 Section 2

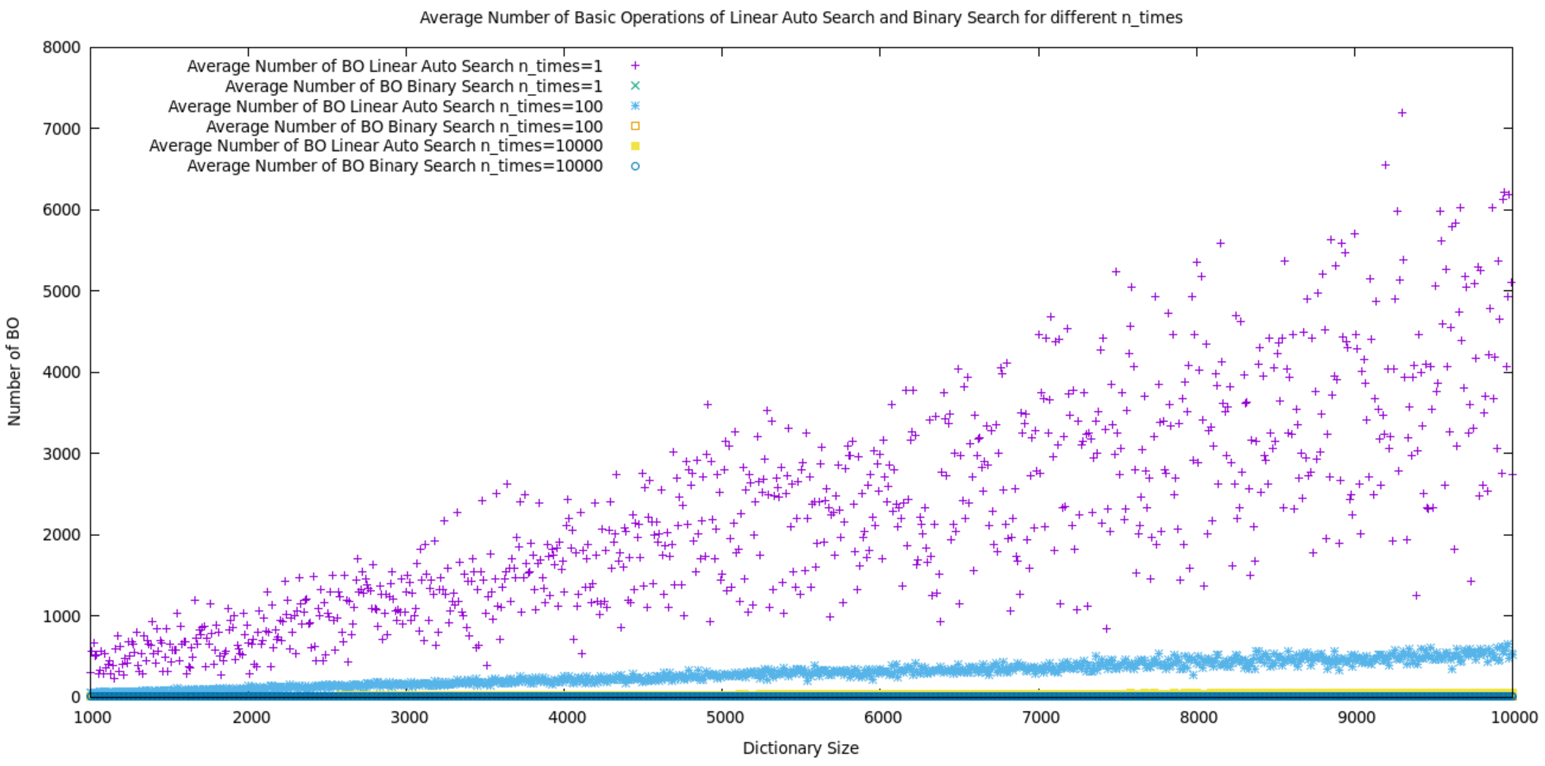
In this section we have executed the file exercise2.c. We have obtained data for dictionary sizes between 1000 and 10000 elements, with increments of size 10. The number of times of each search depends on the plot. For the first two, it is 1, while on the rest it is indicated on the legend.

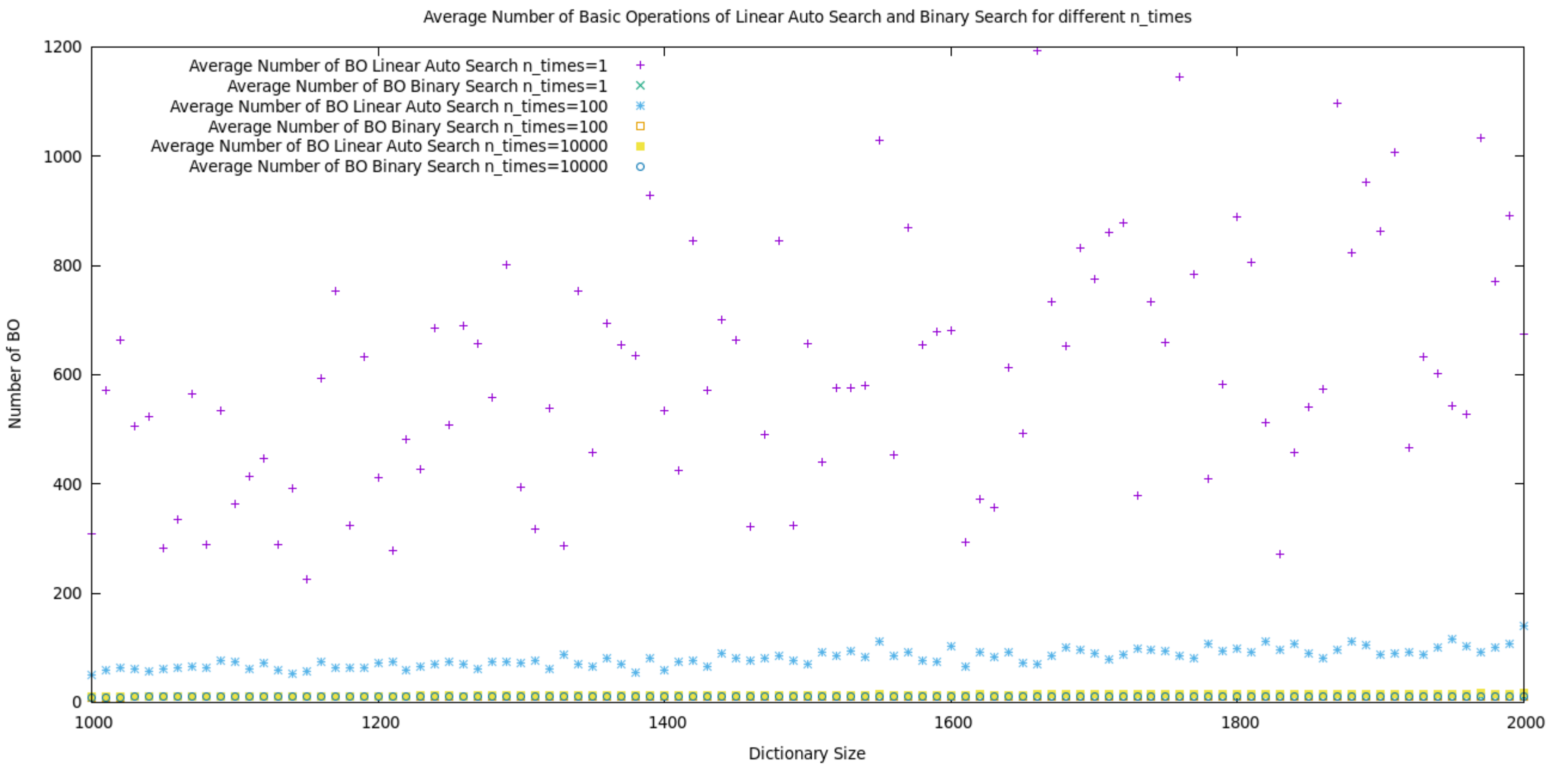


Plot comparing the average number of BOs of linear and binary search approaches. In the x axis the size of the dictionary is represented, and in the y axis the number of BOs is represented. In the plot we can see that the Average Number of BO for Binary Search is in the order of (alogx + b), where the logarithm is in base 2, and (ax+b) in the case of Linear Search. We can see that they seem to be coherent with the theoretical predictions.

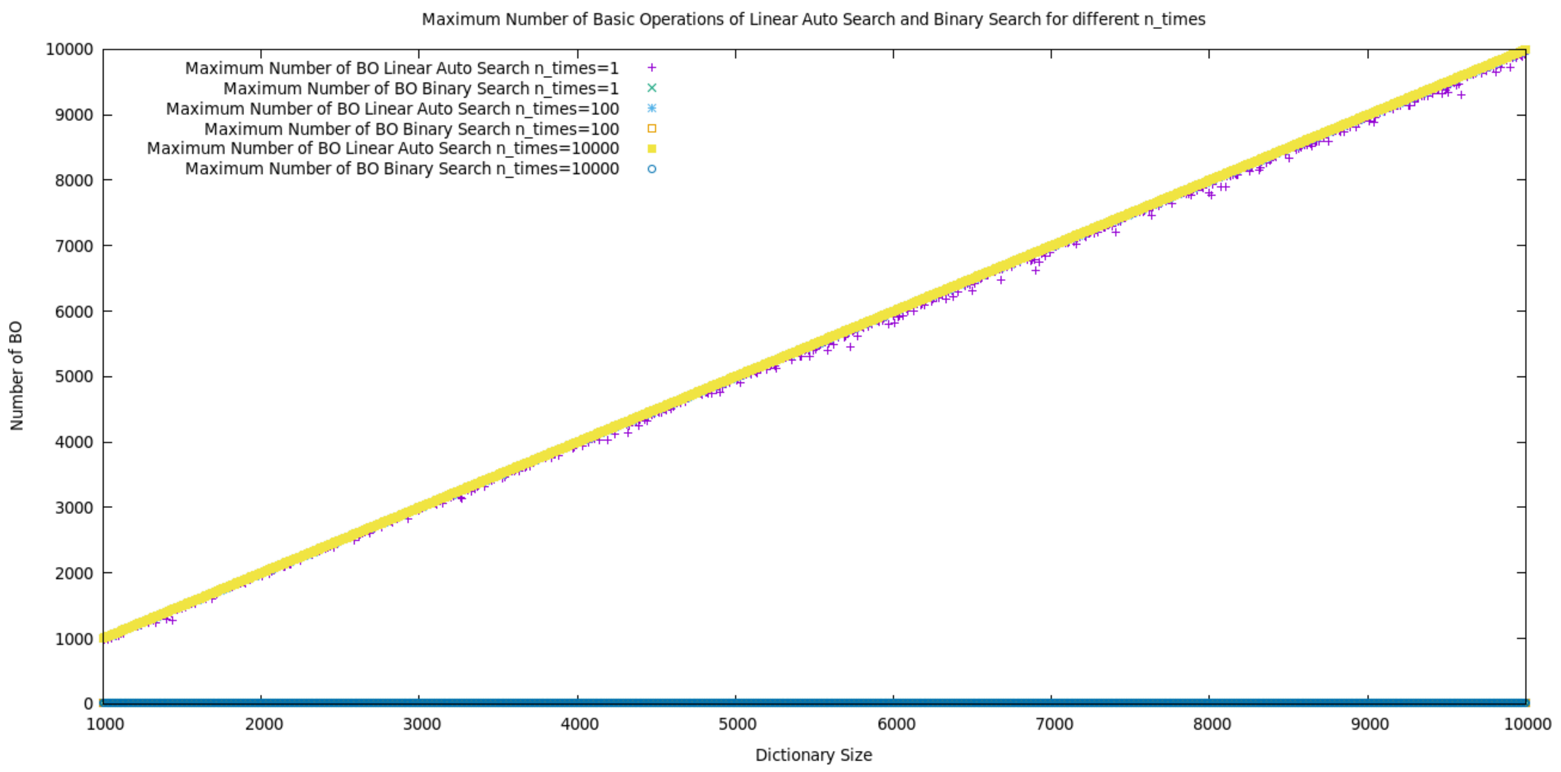


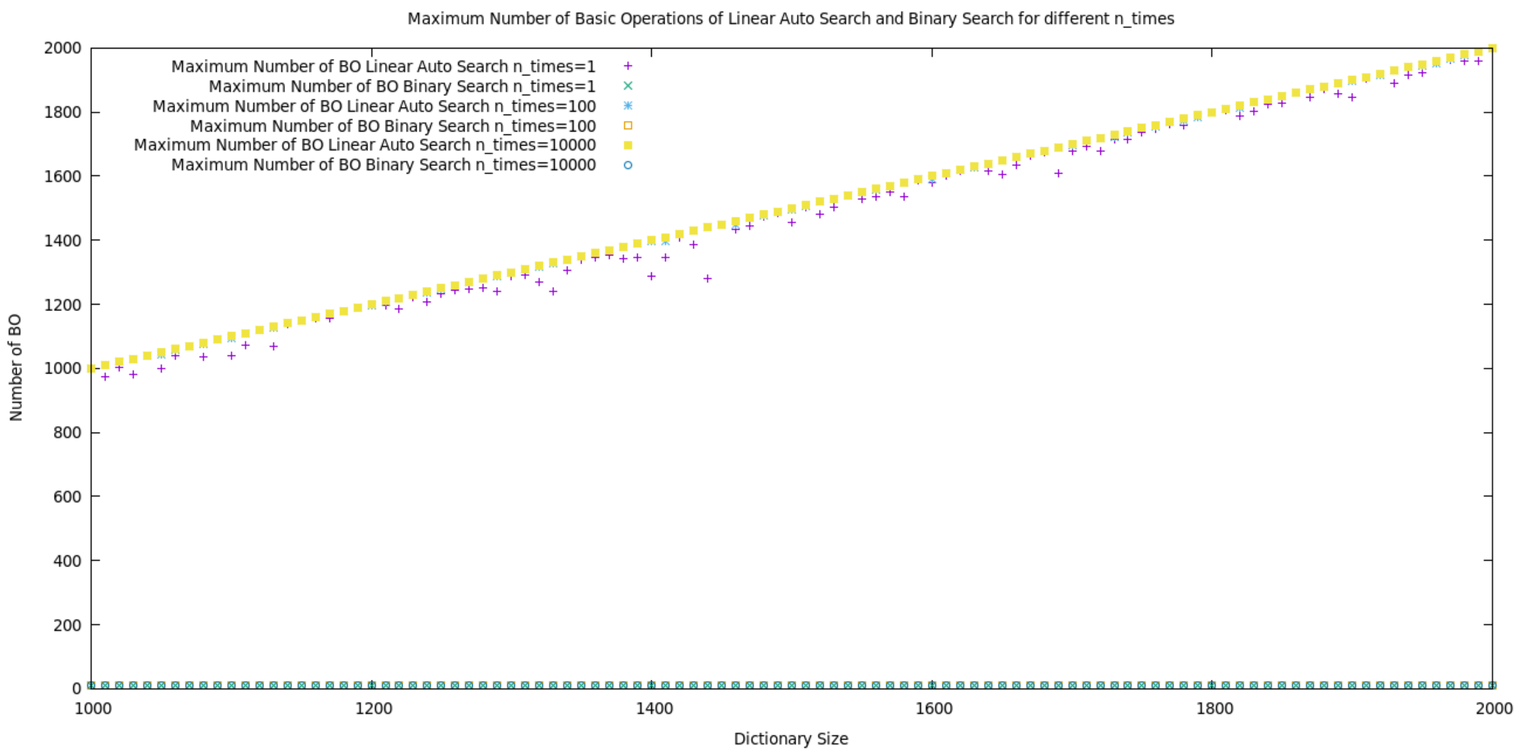
Plot comparing the average clock time for the linear and binary search approaches. In the x axis the size of the dictionary is represented, and in the y the average time in seconds. In the plot we can see that the Average Time for Binary Search is in the order of (alogx + b), where the logarithm is in base 2, and (ax+b) in the case of Linear Search. We can see that they seem to be coherent with the theoretical predictions.



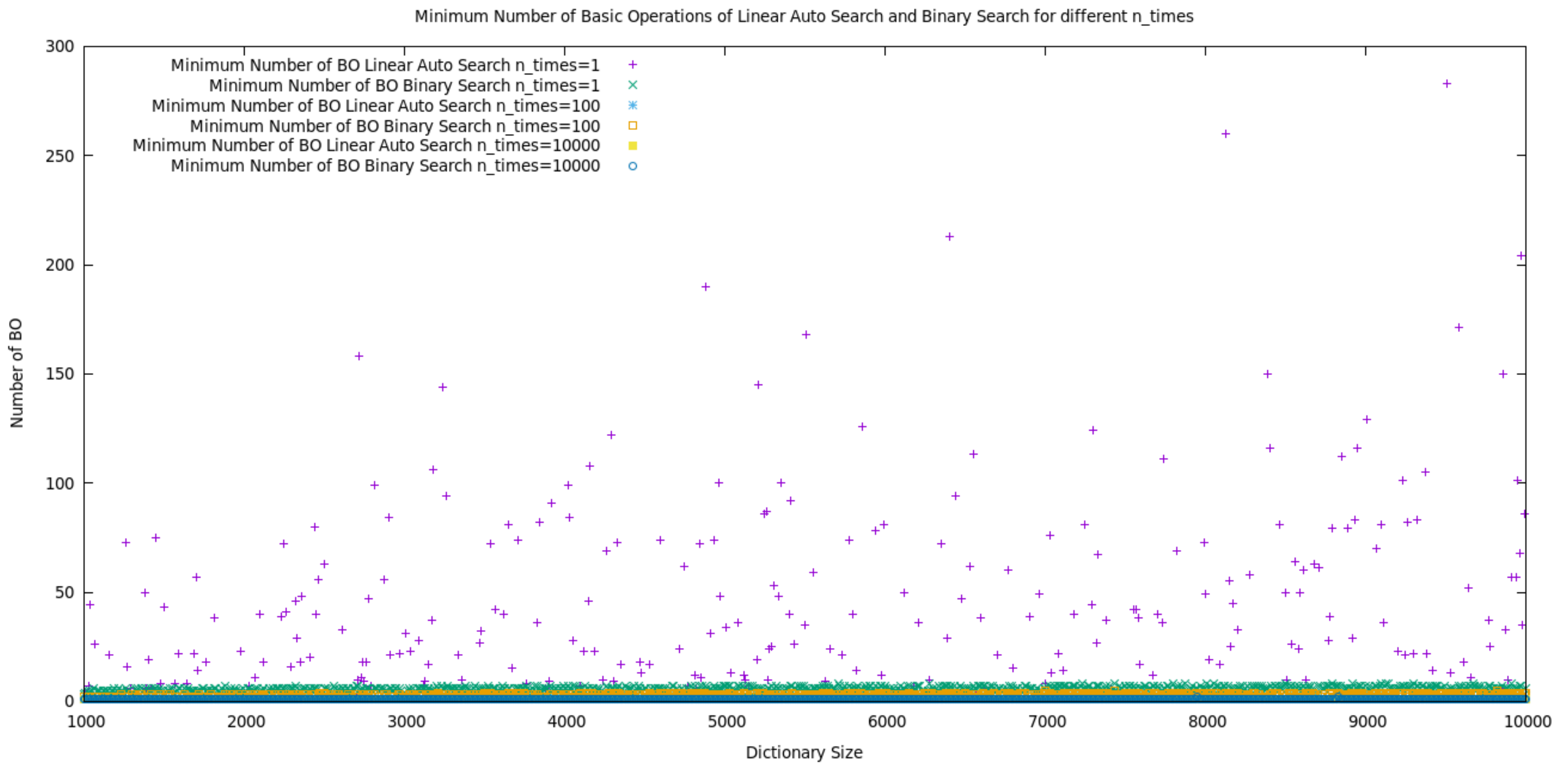


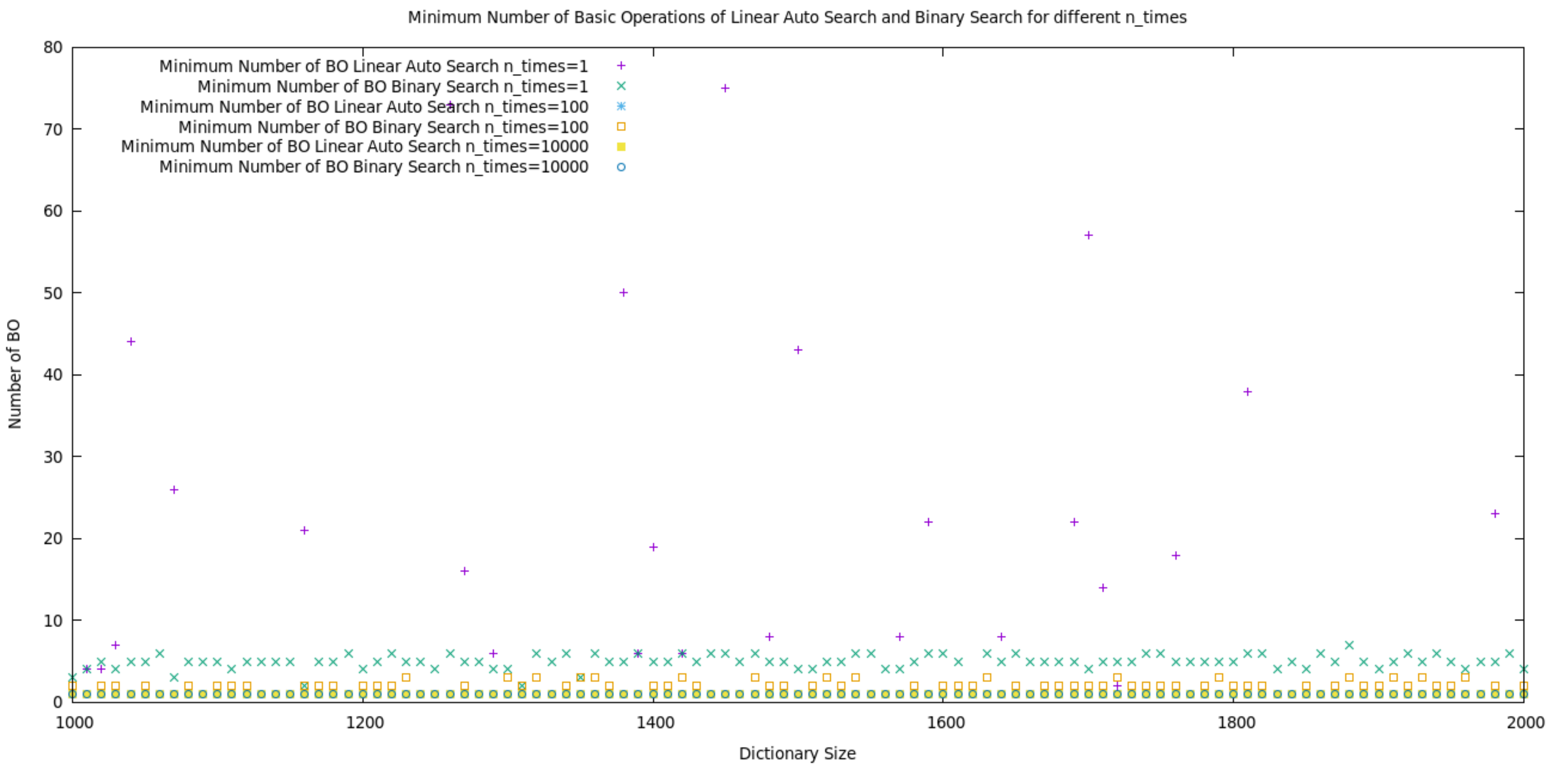
Plot comparing the average number of BOs of the binary and auto-organized linear search (for n_times=1, 100 y 10000). In the x axis the size of the dictionary is represented, and in the y axis the number of BOs is represented. In the plot we can see that the while we increase the number of times that Linear Auto Search is executed, we obtain better results for the average case each time, this is due to the auto-organized linear search property, that leads to a better runtime each time. The second image is a zoomed plot where we can see dictionary sizes from 1000 to 2000 and their average number of BO.



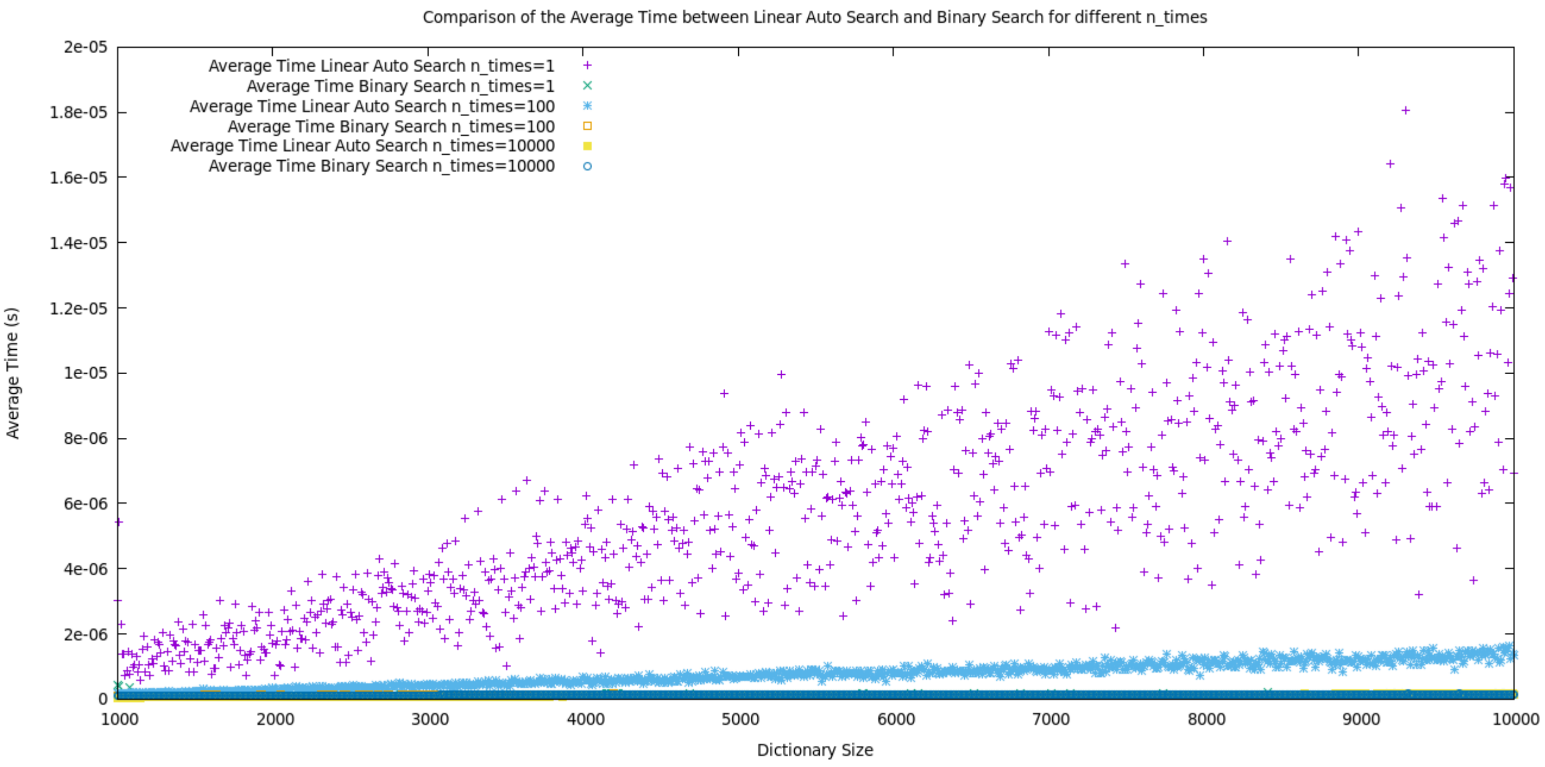


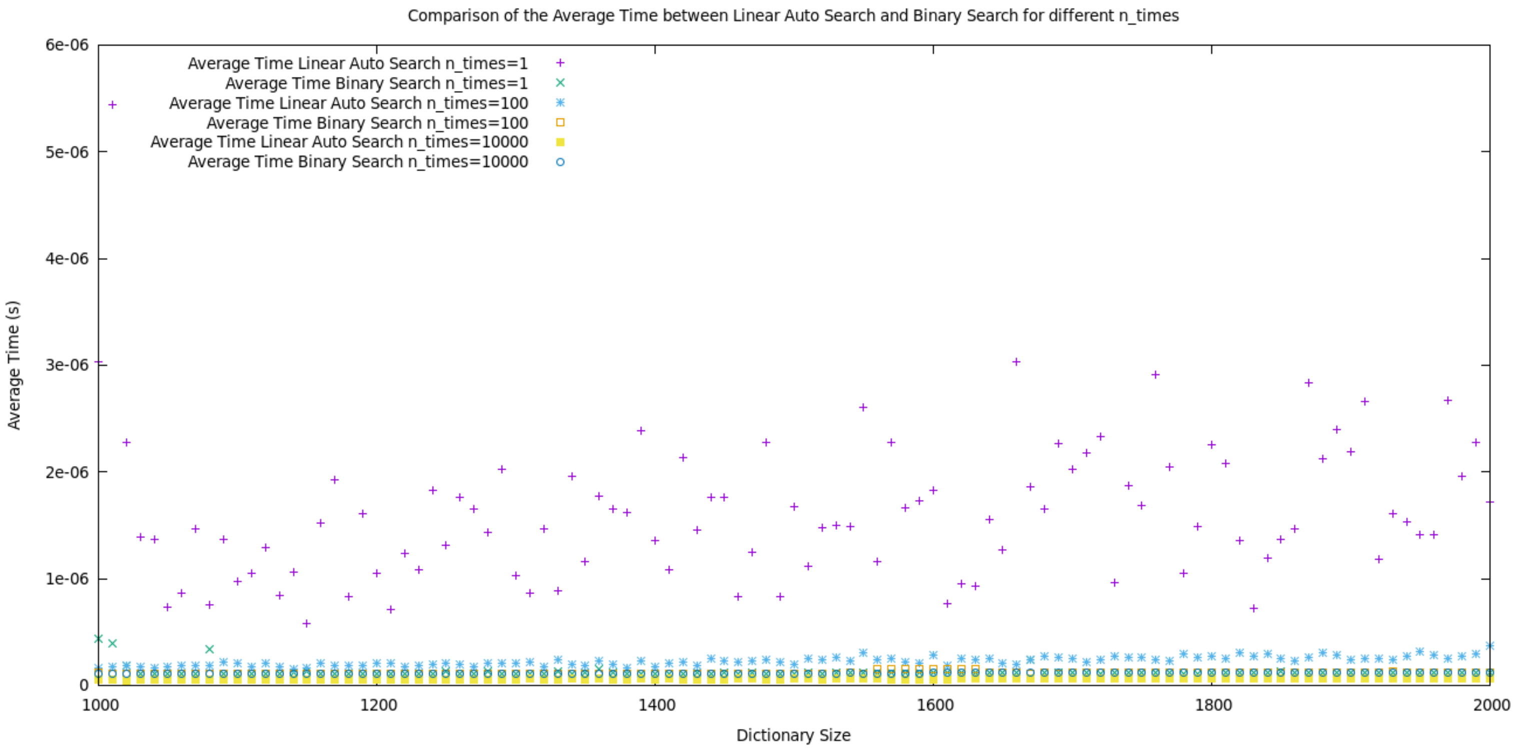
Plot comparing the maximum number of BOs of the binary and auto-organized linear search (for n_times=1, 100 y 10000). In the x axis the size of the dictionary is represented, and in the y axis the number of BOs is represented. In the plot we can see that the while we increase the number of times that Linear Auto Search is executed, we basically obtain a Linear search in the Maximum Number of BOs case. The second image is a zoomed plot where we can see dictionary sizes from 1000 to 2000 and their maximum number of BO.





Plot comparing the minimum number of BOs of the binary and auto-organized linear search (for n_times=1, 100 y 10000). In the x axis the size of the dictionary is represented, and in the y axis the number of BOs is represented. In the plot we can see that the while we increase the number of times that Linear Auto Search is executed, we are getting closer to an O(1) runtime due to the auto-organized linear search property in the Minimum Number of BOs case. the second image is a zoomed plot where we can see dictionary sizes from 1000 to 2000 and their minimum number of BO.





Plot comparing the average clock time for the binary and auto-organized linear search (for n_times=1, 100 y 10000). In the x axis the size of the dictionary is represented, and in the y the average time in seconds. In the plot we can see that the while we increase the number of times that Linear Auto Search is executed, we obtain better results for the average time each iteration, this is due to the auto-organized linear search property, that leads to a better runtime each time.

6. Response to the theoretical questions.

Here, you answer the theoretical questions.

6.1 The basic operation of all of them is the key comparisons that they perform in order to get the element that they are searching for. In linear search and linear auto search is if(table[i]==key) and in binary search: if(table[mid]==key) … if(table[mid]>key)… else

6.2 Binary Search has a Best execution time of O(1) and a worst of O(lgN), on the other hand, Linear Search has a Best execution time of O(1) and a worst of O(N).

6.3 Basically, as we have a potential key distribution rather than an uniform distribution, while we increase the number of times to search, the dictionary gets progressively sorted. In addition, we will be searching more times the elements in the firsts positions than the ones in the last positions, as the probability of searching a small number is greater than a big one.

6.4 The average execution time of linear auto search should be of the order of O(1) when the list is stable, as the probability of searching a small number is greater, and those are at the beginning of the dictionary.

6.5 Binary search uses the technique of "divide and conquer" thus, it gets rid of half of the dictionary in each stage, that is why it has a runtime of lgN. It only works on sorted dictionaries because of how its searching method works. Binary Search compares its target element to the element in the middle of the dictionary, and if it is smaller than it, it searches on the left half of the dictionary recursively, and if it is greater it searches on the right half recursively. This can be done in this way because the dictionary is already sorted.

7. Conclusions.

Working on this practice, we have understand more deeply how searching algorithms work, in particular Linear Auto Search, Binary Search and Linear Search. In addition we have learnt how dictionaries are created, deleted and fulfilled.

We have had the opportunity to put in practice what we have learnt in class, and therefore reinforcing our learning process.

Also we had to overcome some difficulties as coding errors or plots, that had lead us to a better understanding of the tools that we used to accomplish this practice.

After analyzing the results of our algorithms, we had to figure out why some of them were not showing as expected, and finally we came across a logical conclusion.

To sum up, we have demonstrated the theoretical calculus that we made in class by working on this practice, and we have improved our skills on Gnuplot, as well as overcoming some problems that appeared during the process.