Analysis of Algorithms 2020/2021

Practice 3

Kevin de la Coba Malam and Marcos Bernuy, 1391.

Code	Plots	Documentation	Total

1. Introduction

The main goal of this assignment is to implement searching algorithms such as *linear search* and *binary search* using the programming language C. We also have to implement a *linear search* "variation", which whenever the key is found, it is swapped with the previous element.

Once the algorithms are implemented, we have to create functions which are going to test the performance of these algorithms and store the data related with this performance.

The keys are going to be searched in dictionaries, this is going to be the data structure used in order to store our tables. We will implement some primitives to manage the content and the memory usage of the dictionary.

We are also given 2 generators which generates keys. This is going to be useful for having specific keys to search, but these 2 generators doesn't work in the same way, the first one *uniform key generator* generates all the keys given a range, for example if we give to the generator a number N, the generator will fill an array with the numbers from 1 to N, n_times times. The second generator *potential key generator*, creates keys in way which the smaller ones are more likely to appear than the bigger ones.

Once we execute make in the terminal we would have the executables for exercise 1 and 2. In order to execute the first exercise we have to introduce this in the terminal:

```
./exercise1 -size <size> -key <key>
```

We just have to introduce the size and the key we want to look for, it will output something like this:

```
Pratice number 3, section 1
Done by: Kevin de la Coba and Marcos Bernuy
Group: 1292
Search done with linear search:
Key 12 found in position 7278 in 7279 basic op.

Search done with auto-linear search:
Key 12 found in position 7278 in 7279 basic op.

Search done with binary search:
Key 12 found in position 11 in 13 basic op.
```

*We modified the exercise1.c, so now it uses all algorithms.

For the exercise 2 we have to introduce this:

```
./exercise2 -num_min <mínimum size> -num_max <maximum size> -incr <incremento>
-n_times <number of times> -outputFile <file name>
```

After executing the previous command, we will have of the information of the performance stored in the *outputFile* and the following output:

```
Running exercise2
Practice number 3, section 2
Done by: Kevin de la Coba and Marcos Bernuy
Group: 1292
Correct output
```

2. Objectives

2.1 Implementation of the dictionary and the searching algorithms (search.c).

2.1.1 Dictionary

For this section we have to build the primitives for a given data structure:

```
typedef struct dictionary {
  int size; /* table size */
  int n_data; /* number of entries in the table */
  char order;  /* sorted or unsorted table */
  int *table;  /* data table */
} DICT, *PDICT;
```

This is the data structure, it contains a table, the size of the table, the number of elements in the table and the order of the table (sorted or not sorted).

We also have to implement the following primitives:

```
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);
```

With these primitives we will manage the content of each dictionary.

2.1.2 Searching algorithms

Now that we have the structure implemented, we need to implement the different searching algorithms, but first we need to create a generic function to search:

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

With this function we can search in a dictionary with the searching function we want.

```
  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);
```

bin_search refers to the binary search algorithm, **lin_search** refers to the linear search algorithm, and **lin_auto_search** refers to a variation of the linear search algorithm which whenever the key is found it is being swapped with the previous element.

2.2 Comparison of search efficiency (times.c)

In this section we have to implement 2 new functions which will measure the efficiency of the previously mentioned searching algorithms. These are the 2 functions:

```
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);
short average_search_time(pfunc_search metodo, pfunc_key_generator generator,
                          int order,
                          int N,
                          int n_times,
                          PTIME_AA ptime);
```

generate_search_times calls **average_search_time** with the sizes starting from *num_min* to *num_max* with an increment of *incr*. In order to store all the information, we will call the function *save_time_table* which was created in the assignment 1.

These 2 functions are very similar to the ones implemented in the assignment 1.

3. Tools and methodology

We implemented and tested the code in Linux, Ubuntu. We used visual studio code for implementing the code in C, for testing it and debugging it too.

3.1 Implementation of the dictionary and the searching algorithms (search.c).

In this part of the assignment we implemented the code related to the dictionary structure.

3.1.1 Dictionary

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

  In this function we just have to initialize the dictionary. For this we first allocate memory for one dictionary, then we initialize the variables of the structure, and finally, given a size specified as an argument, we allocate memory inside of this dictionary for a table.

- `void free_dictionary(PDICT pdict);`

  We just simply free the memory of the table and the dictionary itself.

- `int insert_dictionary(PDICT pdict, int key);`

  Given a key and a dictionary we insert the key in the dictionary but, depending on the type of the dictionary (sorted or not sorted), we just simply place it at the end, or we place it in his corresponding position with the algorithm shown in the statement.

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

Given a dictionary, an array of keys and the number of keys in that given dictionary, we created a loop which iterates through the keys and inside this loop we call the `insert_dictionary` function to insert each key of the array.

2.1.2 Searching algorithms

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

We created this function as a "generic" function which searches for a key in a given dictionary using the given "method", which can be *linear search*, *binary search*… When the key is found, `*ppos` contains the position where it was found.

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

This is the binary search algorithm, for this algorithm we followed the theory slides.

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

This is the linear search algorithm, for this algorithm we just followed the theory slides.

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

This is algorithm is very similar to `lin_search` but, the difference is that whenever a key is founded we do a swap with the previous element, so the table is modified.

2.2 Comparison of search efficiency (times.c)

Here we implemented 2 functions for searching a lot of keys.

- ```
  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);
  ```

In this function we just call the next function `average_search_time` several times (from num_min to num_max, incrementing incr sizes), we create the PTIME structure and after it is being filled with data, use `save_time_table` to save the data into a file.

- ```
  short average_search_time(pfunc_search metodo,
  pfunc_key_generator generator,
            int order,
            int N,
            int n_times,
            PTIME_AA ptime);
  ```

In this function we first create a dictionary, then we fill it with a permutation, we create all the keys we are going to look for and then we

search them, while we search them, we measure the time and finally we fill all the measure data in the PTIME structure.

4. Source code

4.1 Implementation of the dictionary and the searching algorithms (search.c)

4.1.1 Dictionary

```
/**
 * @brief Function to initialize the dictionary, reserves memory for it
 *
 * @param size Size of the dictionary
 * @param order Order of the table. Ordered or not ordered
 * @return PDICT Created dictionary
 */
PDICT init_dictionary(int size, char order) {

    PDICT pdict = NULL;

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

    pdict = (PDICT)malloc(sizeof(pdict[0]));
    if (pdict == NULL)
        return NULL;

    /* Initializing the variables */
    pdict->n_data = 0;
    pdict->size = size;
    pdict->order = order;

    /* Reserving memory for the table */
    pdict->table = (int *)calloc(size, sizeof(pdict->table[0]));
    if (pdict->table == NULL) {
        free(pdict);
        return NULL;
    }

    return pdict;
}

/**
 * @brief Function to free the resources of the dictionary
 *
 * @param pdict Dictionary to free
 */
void free_dictionary(PDICT pdict) {
    if (pdict == NULL) return;

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

/**
```

```c
 * @brief Function to insert a key into a dictionary
 *
 * @param pdict Dictionary where the key is going to be inserted
 * @param key Key to insert
 * @return int Basic operations done
 */
int insert_dictionary(PDICT pdict, int key) {

    int j = 0, BO = 0;

    if (pdict == NULL || key <= 0) return ERR;

    /* Checking how the table is sorted and if it is full */
    if (pdict->order == NOT_SORTED && pdict->n_data < pdict->size) {

        pdict->table[pdict->n_data] = key;
        pdict->n_data += 1;

        /* We didn't do any basic operation BO=0 */
        return BO;
    } else if (pdict->order == SORTED && pdict->n_data < pdict->size) {

        pdict->table[pdict->n_data] = key;
        j = pdict->n_data - 1;

        while (j >= 0) {
            if (pdict->table[j] > key) {
                pdict->table[j+1] = pdict->table[j];
                j--;
            } else break;
            BO++;
        }
        pdict->table[j+1] = key;
        pdict->n_data += 1;

        return BO;
    }

    return ERR;
}

/**
 * @brief Function to insert several keys into a dictionary
 *
 * @param pdict Dictionary where the keys are going to be inserted
 * @param keys Keys to be inserted
 * @param n_keys Number of keys to be inserted
 * @return int Basic operations done
 */
int massive_insertion_dictionary(PDICT pdict, int *keys, int n_keys) {

    int i = 0, BO = 0, ret = 0;

    if (pdict == NULL || keys == NULL || n_keys <= 0) return ERR;

    /* Checking if there's enough space */
    if (pdict->size < (pdict->n_data + n_keys)) return ERR;
```

```
    /* Inserting keys */
    for (i = 0; i < n_keys; i++) {

        if (keys[i] <= 0) return ERR;
        ret = insert_dictionary(pdict, keys[i]);
        if (ret == ERR) return ERR;
        BO += ret;
    }

    return BO;
}
```

4.1.2 Searching algorithms

```
/**
 * @brief Function to search in the dictionary a key given a function
 *
 * @param pdict Dictionary where we are going to search the key
 * @param key Key to search
 * @param ppos Variable passed by reference where we are going
 *             to store the index of the key
 * @param method Method to use for searching
 * @return int Basic operations done, ERR in case of error and NOT_FOUND if we
don't find the key
 */
int search_dictionary(PDICT pdict, int key, int *ppos, pfunc_search method) {

    int BO = 0;

    /* Checking arguments */
    if (pdict == NULL || key <= 0 || ppos == NULL || method == NULL)
        return ERR;

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

    /* In case we didn't find the key */
    if (*ppos == NOT_FOUND) return NOT_FOUND;

    return BO;
}

/* Search functions of the Dictionary ADT */
/**
 * @brief Function that searches into a table using the
 *        binary search algorithm
 *
 * @param table Table where we have to search
 * @param F First index of the table
 * @param L Last index of the table
 * @param key Key to search
```

```c
 * @param ppos Variable passed by reference where we are going
 *             to store the index of the key
 * @return int Basic operations done
 */
int bin_search(int *table, int F, int L, int key, int *ppos) {

    int middle = 0, BO = 0, start = 0, end = 0;

    /* Checking arguments */
    if(table == NULL || F < 0 || L < 0 || F > L || key <= 0 || ppos == NULL)
        return ERR;

    start = F;
    end = L;
    while(start <= end) {
        middle = (start + end)/2;
        BO++;
        if(key == table[middle]) {
            *ppos = middle;
            return BO;
        } else if (key < table[middle]) {
            end = middle - 1;
        } else {
            start = middle + 1;
        }
    }
    *ppos = NOT_FOUND;
    return BO;
}

/**
 * @brief Function that searches into a table using the
 *        linear search algorithm
 *
 * @param table Table where we have to search
 * @param F First index of the table
 * @param L Last index of the table
 * @param key Key to search
 * @param ppos Variable passed by reference where we are going
 *             to store the index of the key
 * @return int Basic operations done
 */
int lin_search(int *table, int F, int L, int key, int *ppos) {

    int i = 0, BO = 0;

    if (table == NULL || F < 0 || L < 0 || L < F || key <= 0 || ppos == NULL)
        return ERR;

    /* Iterating through all the elements of the array */
    for (i = 0; i <= L; i++) {
        BO++;
        /* Key found, saving index */
        if (table[i] == key) {
            *ppos = i;
            return BO;
        }
```

```
    }

    /* If we reach this point it means that we didn't find the key */
    *ppos = NOT_FOUND;
    return BO;
}

/**
 * @brief Function that searches a key as its done in the linear search
 *        algorithm but when the key is founded we swap it with the previous
one
 *
 * @param table Table where we have to search
 * @param F First index of the table
 * @param L Last index of the table
 * @param key Key to search
 * @param ppos Variable passed by reference where we are going
 *             to store the index of the key
 * @return int Basic operations done
 */
int lin_auto_search(int *table, int F, int L, int key, int *ppos) {

    int i = 0, BO = 0;

    if (table == NULL || F < 0 || L < 0 || L < F || key <= 0 || ppos == NULL)
        return ERR;

    /* Iterating through all the elements of the array */
    for (i = 0; i <= L; i++) {
        BO++;

        /* Key found, saving index and swapping*/
        if (table[i] == key) {
            *ppos = i;

            /* Swapping */
            if (i > F) swap(&table[i], &table[i-1]);
            *ppos = i;
            return BO;
        }
    }

    /* If we reach this point it means that we didn't find the key */
    *ppos = NOT_FOUND;
    return BO;
}
```

4.2 Comparison of search efficiency (times.c)

```
/**
 * @brief Given a method to search, the function searches keys generated by a
given generator
 *        The data related with the performance of the searching is stored in
a given file
 *
```

```
 * @param method Function to use in order to search keys
 * @param generator Generator which generates keys
 * @param order Order of the table. It can be ordered or dissordered
 * @param file File name where the data related with the performance is going
to be stored
 * @param num_min Minumum size of the tables
 * @param num_max Maximum size of the table
 * @param incr Increment in the table size per iteration
 * @param n_times Number of times to search in each table
 * @return short Status of the function
 */
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) {
    int i = 0, size = 0, n = 0;
    PTIME_AA ptime = NULL;

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

    /* Allocating memory for the array of ptimes */
    size = ((num_max - num_min) / incr) + 1;
    ptime = (PTIME_AA)malloc(size * sizeof(ptime[0]));
    if (ptime == NULL) return ERR;

    /* Getting the average searching time for each size */
    for (i = num_min; i <= num_max; i += incr, n++) {
        if (average_search_time(method, generator, order, i, n_times,
&ptime[n]) == ERR) {
            free(ptime);
            ptime = NULL;
            return ERR;
        }
    }

    /* Saving the times in a file */
    if (save_time_table(file, ptime, size) == ERR) {
        free(ptime);
        ptime = NULL;
        return ERR;
    }

    free(ptime);
    ptime = NULL;
    return OK;
}

/**
 * @brief Given a size, a method and a key generator, the function searches in
a table
 *        the generated keys n_times times
 *
 * @param method Function to use in order to search keys
```

```
 * @param generator Generator which generates keys
 * @param order Order the table
 * @param N Size of the table
 * @param n_times Number of times to search in each table
 * @param ptime Time structure to store the data related with the performance
 * @return short Status of the function
 */
short average_search_time(pfunc_search metodo, pfunc_key_generator generator,
                          int order,
                          int N,
                          int n_times,
                          PTIME_AA ptime) {

    double total = 0;
    int *perm = NULL, *keys = NULL, i = 0, ppos = 0;
    long BO = 0, total_BO = 0, mul = 0;
    PDICT dict = NULL;
    struct timeval  tv1, tv2;

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

    mul = (long)(N*n_times);

    /* Creating a dictionary */
    dict = init_dictionary(N, (char)order);
    if (dict == NULL) return ERR;

    /* Creating a permutation */
    perm = generate_perm(N);
    if (perm == NULL) {
        free_dictionary(dict);
        return ERR;
    }

    /* Inserting the elements of the permutation into the dictionary */
    total_BO = massive_insertion_dictionary(dict, perm, N);
    if (total_BO == ERR) {
        free(perm);
        perm = NULL;
        free_dictionary(dict);
        return ERR;
    }

    /* Creating an array for the keys */
    keys = (int*)malloc(mul*sizeof(keys[0]));
    if (keys == NULL) {
        free(perm);
        perm = NULL;
        free_dictionary(dict);
        return ERR;
    }

    /* Generating keys */
    generator(keys, mul, N);
```

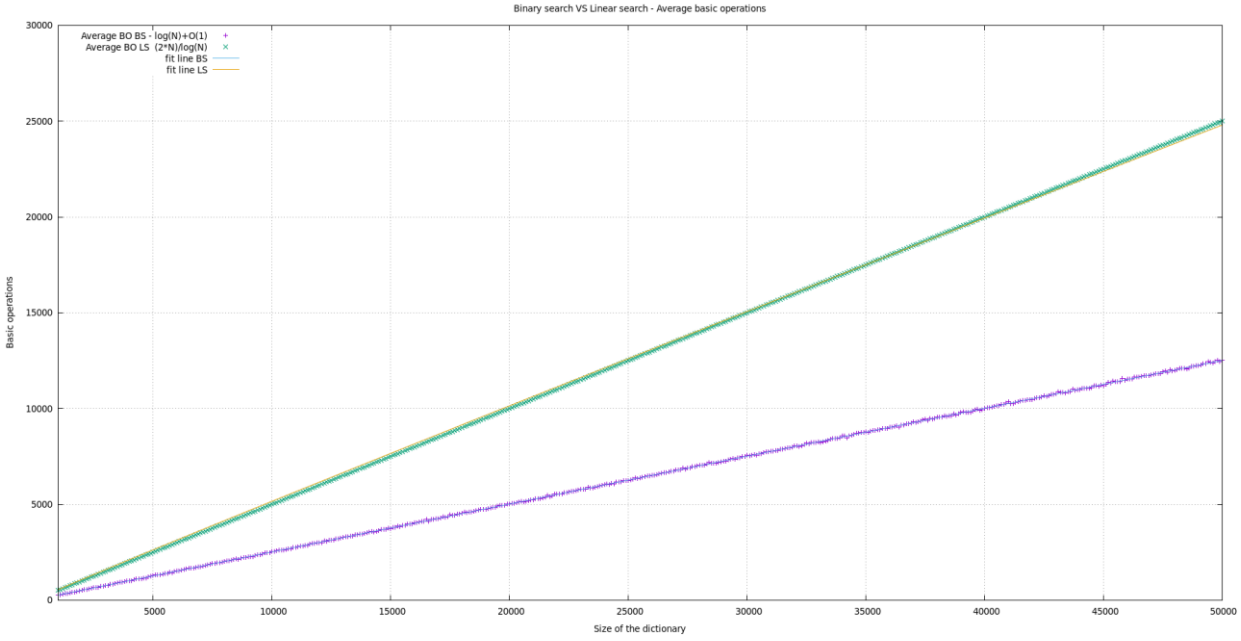
```
    /* Initializing ptime values */
    ptime->min_ob = __INT_MAX__;
    ptime->max_ob = 0;

    if (gettimeofday(&tv1, NULL) != 0) {
        free(perm);
        free(keys);
        keys = NULL;
        perm = NULL;
        free_dictionary(dict);
        return ERR;
    }

    /* Searching for the keys */
    for (i = 0; i < mul; i++) {
        BO = search_dictionary(dict, keys[i], &ppos, metodo);
        if (BO == ERR || BO == NOT_FOUND) {
            free(perm);
            free(keys);
            keys = NULL;
            perm = NULL;
            free_dictionary(dict);
            return ERR;
        }
        if (ptime->min_ob > BO) ptime->min_ob = BO;
        if (ptime->max_ob < BO) ptime->max_ob = BO;
        total_BO += BO;
    }

    if (gettimeofday(&tv2, NULL) != 0) {
        free(perm);
        free(keys);
        keys = NULL;
        perm = NULL;
        free_dictionary(dict);
        return ERR;
    }

    /* Assingning the values to the structure */
    total = (double) (tv2.tv_usec - tv1.tv_usec) / 1000000 + (double)
(tv2.tv_sec - tv1.tv_sec);
    ptime->time = total / (double) (mul);
    ptime->N = N;
    ptime->average_ob = (double)(total_BO/(double)mul);
    ptime->n_elems = mul;

    free_dictionary(dict);
    free(perm);
    free(keys);
    keys = NULL;
    perm = NULL;
    return OK;
}
```

5. Results, plots

5.1 Section 1

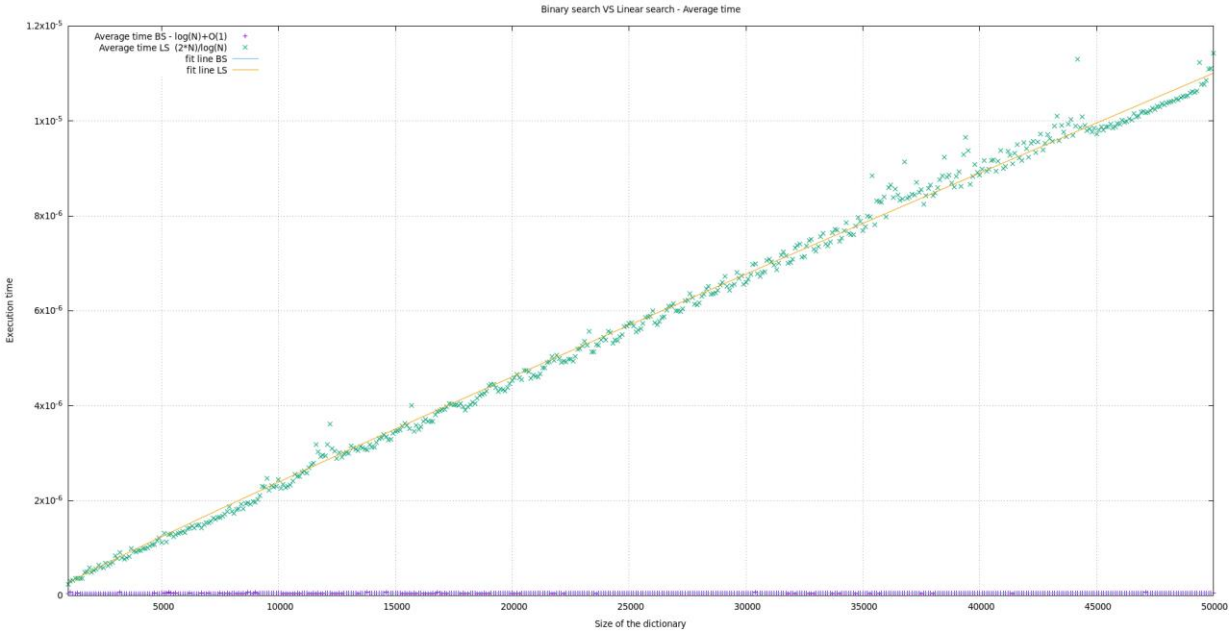
```
./exercise1 -size 15000 -key 12

Running exercise1
Pratice number 3, section 1
Done by: Kevin de la Coba and Marcos Bernuy
Group: 1292

Search done with linear search:
Key 12 found in position 8989 in 8990 basic op.

Search done with auto-linear search:
Key 12 found in position 8989 in 8990 basic op.

Search done with binary search:
Key 12 found in position 11 in 13 basic op.
```

5.2 Section 2



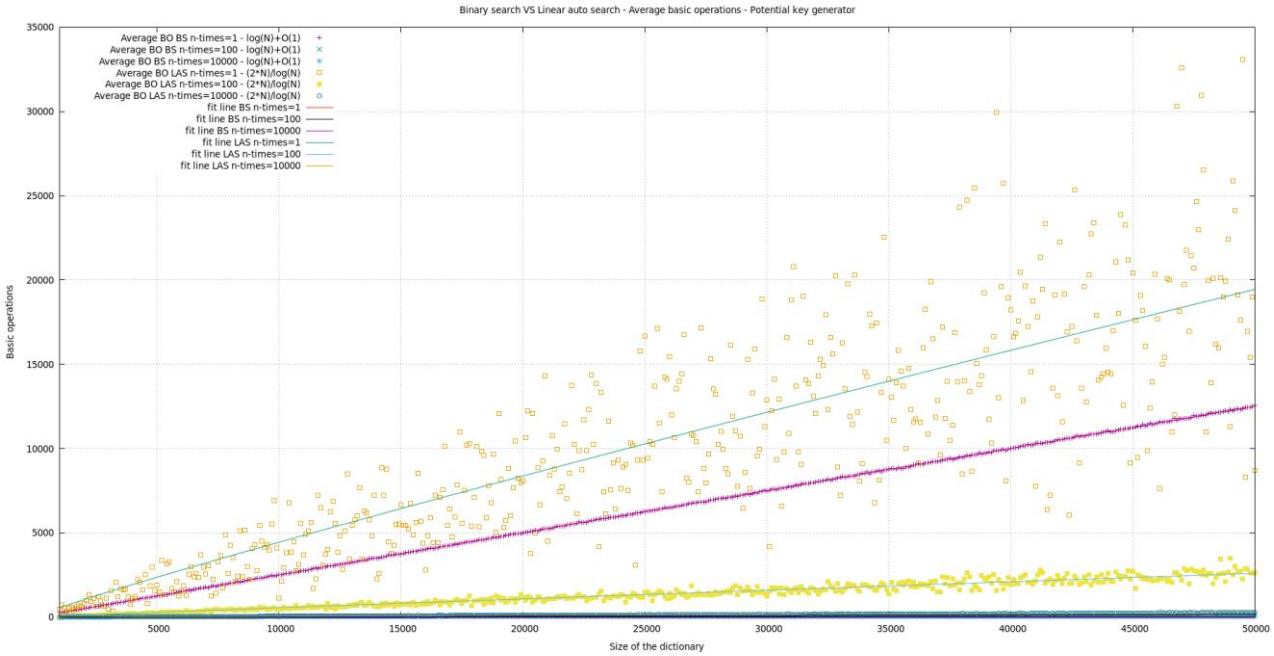
In this plot we can see the performance of both functions, *binary search* and *linear search*. As expected, *linear search* has a worse performance than *binary search*.

The way this is measured is by searching keys in a dictionary, adding all the basic operations done and finally dividing them by the number of keys searched. We can say that is not a “realistic” average regarding the performance of searching a key, it is the average regarding searching multiple keys.



In this second plot we can see the average time of each of the functions. We can appreciate the huge gap that there is between the functions. *Binary search* has a better performance than linear search.

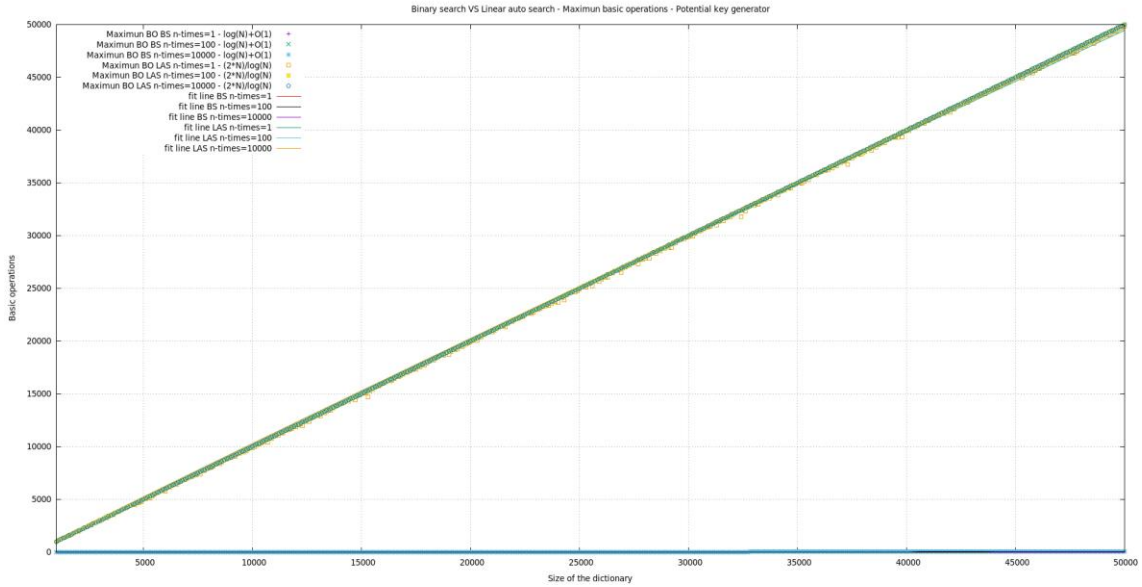
If we divide the last data of the *linear search* and the last data of the *binary search* 1.142518e-05/5.518000e-08 = 422688, we can see that binary search in table of 50.000 elements is 422688 times faster. We can conclude then that *binary search* is way faster than *linear search*.



This is the first plot where we can observe the *linear auto search* behavior.

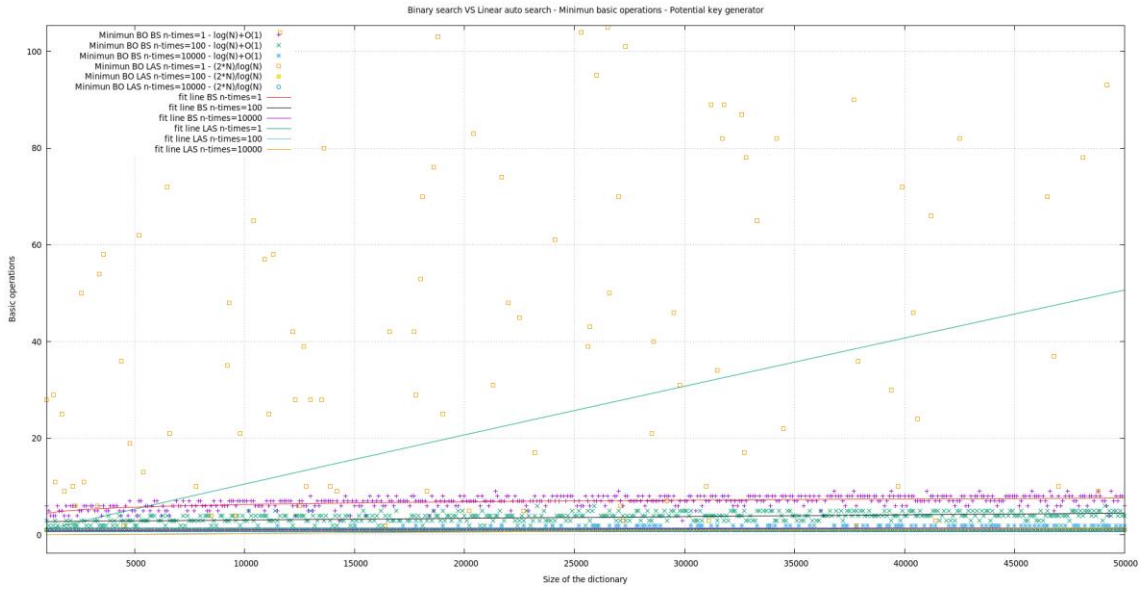
If we visualize the plot we can realize that the bigger the size of the dictionary is the more the behavior of the algorithm looks like the *binary search* growth. This is due to the fact

that when using small sizes, the swap made in the algorithm is useless. On the other hand, when the swap is made for sizes which are bigger, then it becomes more useful because it gets to a point in which the key looked for is in the first position and therefore the cost is reduced.



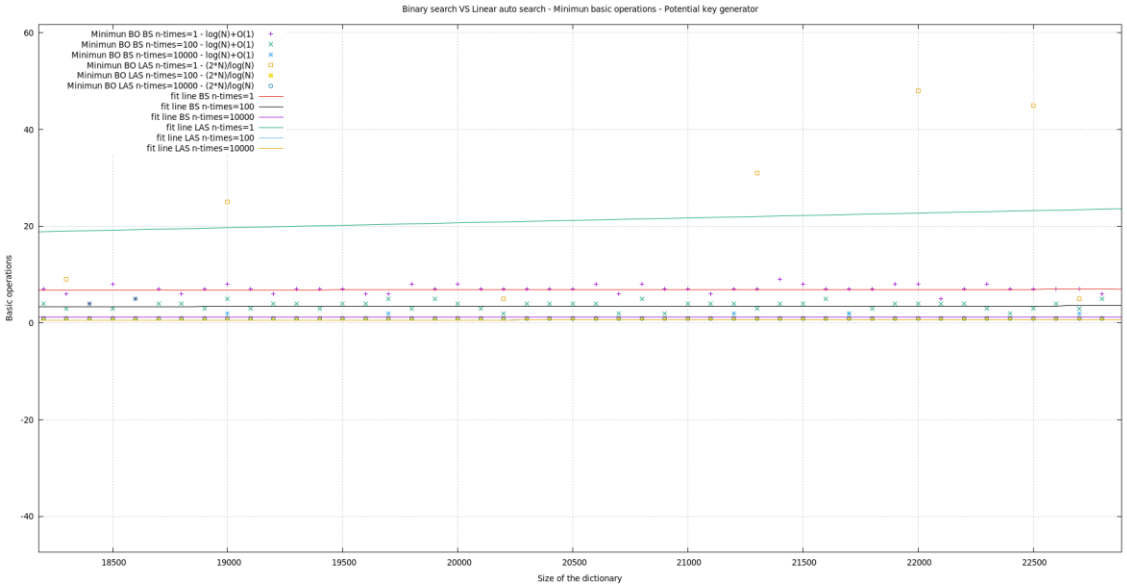
This image shows the huge difference when comparing the maximum basic operations between the *auto-organized linear search* and the *binary search.*

While the binary search is steady and stays at the lowest basic operations even when taking the maximum values, the auto-linear search has a slope which defines the growth of the graph. The maximum values have that slope because even if the percentage of having to find a key that is at the end of the dictionary are low, it's still possible, especially with the first keys being searched for.

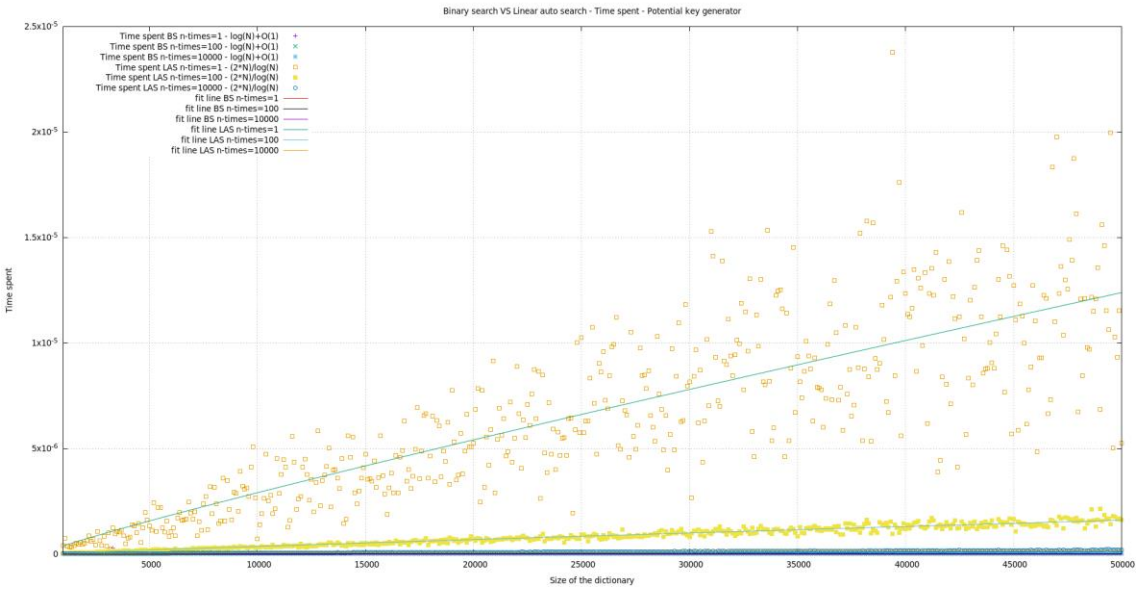


When it comes to the plot when taking the minimum values, the difference between both algorithms decreases but it still stays pretty significant. In this case, in low sizes such as

1, the auto-organized linear search is pretty irregular and therefore it is the least efficient. With higher values the difference is less between them so we decided to zoom on it.



In the zoom we now can see that for the highest size, both algorithms have more or less the same basic operations. As it was explained before, this is because with large sizes the auto-organized linear search tends to become more efficient because the key is in a high rate at the first positions.



In this picture, we can see the difference in time performance between *auto-organized linear search* and *binary search*.

The bigger the size of the dictionary gets the smaller the difference in time performance is, as it was expected.

5. Response to the theoretical questions

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

The comparison between the key and an element of the table.

In linear search and linear auto search, the basic operation is a key comparison between the key and the element in the table traversed at that moment. `if (table[i] == key)`

For binary search the basic operation is a key comparison between the key and the middle element of the table in that iteration of the loop (the middle element changes in every iteration of the loop). `if(key == table[middle])`

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.*

$\boldsymbol{W_{BS}(n) = O(log_2(n))}$ In each iteration of the algorithm we divide the table and we check the middle element of that division, so given a table with size n, the maximum times we can divide the table is $log_2(n)$, so in the last division we would have in the middle of the table (the table length is 1) the element we are looking for.

$\boldsymbol{B_{BS}(n) = O(1)}$ In the case that the key is in the middle of the table.

$\boldsymbol{W_{LS}(n) = O(n)}$ Linear search goes from the beginning of the table to the end, so in the case that the element is in the last position of the table, we would have made n comparisons.

$\boldsymbol{B_{BS}(n) = O(1)}$ In the case that the key is in the first element.

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?*

The potential key generator generates more the smaller keys, so, having a table with numbers going from 1 to N (not sorted), using *lin auto search* and using the potential key generator for generating keys in a range of 1 to N several times, we would see that the after some searching the smaller keys are going to be at the beginning of the table. If we continue searching the table would have the smaller values at the beginning and the bigger ones at the end, this doesn't mean that is going to be sorted.

1 will be the first, it has a probability of 50%. It will be 1st most of the times not always, because 2 will be the second because it is the second most probable number that can be generated (17%), but if 2 is generated then it is swapped with one, and now the 2 is the first element, then 1 could be generated so it will be swapped with 2 and so on …

To sum up, **the smaller numbers are going to be at the beginning**.

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.*

The average execution time function looks like a 1/n where n ≥1. At the beginning the algorithm will struggle to find keys, but once several searches are done, the most likely to search keys will be at the beginning, so the searches will be faster.

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

It works well because it is sorted, and it divides de table.

$Being\ T\ a\ table\ with\ N\ elements\ from\ T(1)\ to\ T(N), we\ know\ that$:

$$T(i-1) < T(i) < T(i+1) | i < N\ \wedge i > 1$$

This means that we look for a key which $k \in T$,

$if\ k < T(\lfloor N/2 \rfloor)$ we would have to look in the first half of the table (T').

In case that $k >\ T(\lfloor N/2 \rfloor)$, we would look in the second half of the table.

$If\ k = T(\lfloor N/2 \rfloor)$ then we would have found the key.

The process of diving the table by 2 in each iteration is very powerful, because we can divide a table by 2 a maximum number of times equal to $\lfloor log_2(N) \rfloor$.

We can check this with an example:

Let us imagine that we have a table of 2.000.000 elements. It is a big table but, how many times do we have to divide it until the result of the division is 1?

2.000.000/2→1.000.000/2→500.000/2→250.000/2→125.000/2→62.500/2→31.250/2 →15.625/2→7812/2→3906/2→1953/2→976/2→488/2→244/2→122/2→61/2→ 30/2→15/2→7/2→3/2→1.

20 = $\log_2$ 2.000.000 divisions we performed until we had one, this will be as if we have the key in the last or first position. The algorithm is very fast looking for keys.

6. Conclusions

In this assignment we implemented searching algorithms. We experienced how divide and conquer algorithms are better, and we could see it when we were measuring the performance in dictionaries of different sizes. Measuring the performance helped us to understand the behavior of both algorithms.

We also understood that there are more options apart from divide and conquer, as we saw in *linear auto search*, having the most searched keys at the beginning makes the

algorithm very efficient, although we have to reach the point where the most searched keys are in the beginning of the table.

To sum up, the normal linear search is the less efficient and the linear auto search is still less efficient than binary search, but it is useful when we perform a lot of searches of the same key.