Analysis of Algorithms 2019/2020

Practice 3

Laura de Paz and Paula Samper, Group 1251

Code	Plots	Documentation	Total

1. Introduction.

In the last practice of this course we have put into practice the knowledge we have acquired in the theoretical lessons during these lectures of this course. For instance, we have implemented a dictionary ADT, on which we later worked with three different search algorithms: linear search, binary search and linear auto search. Then we showed their worst, average and best case as previously studied in class.

2. Objectives

2.1 Section 1

The objective of this session is to implement the dictionary ADT. Firstly we have to create the DICT structure in *search.c* to store all the information of the dictionary.

Then we will implement the following functions in *search.c* to manage the data within the dictionary.

> *PDICT init_dictionary (int size,char order);* → to initialize a dictionary structure.
>
> *void free_dictionary(PDICT pdict);* → to destroy a dictionary.
>
> *int insert_dictionary(PDICT pdict,int key);* → to insert an element into a dictionary.
>
> *int massive_insertion_dictionary (PDICT pdict,int *keys,int n_keys);* → to insert an array of elements into a dictionary.
>
> *int search_dictionary(PDICT pdict,int key,int *ppos,pfunc_search method);* → to search an element within a dictionary, with a given search method.

We will also have to implement the following searching algorithms, also in *search.c*:

> *int bin_search(int *table,int F,int L,int key,int *ppos);* → Binary search
>
> *int lin_search(int *table,int F,int L,int key,int *ppos);* → Linear search
>
> *int lin_auto_search(int *table,int F,int L, int key,int *ppos);* → Auto-organized linear search

All of these functions (except for *init_dictionary*, and *free_dictionary*) will return the number of basic operations, or ERR in case of error. Finally, we will use the file *exercise1.c* to check the correctness of our code. We will change it so we can check the behaviour of the binary and linear search algorithms over a sorted table.

2.2 Section 2

Our goal in this section is to measure the efficiency of the three searching algorithms we have previously implemented. To achieve this, we will add to *times.c* the two following functions:

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

Where *order* indicates if the dictionary is sorted, *N* is the size of the dictionary, *n_times* shows how many times each one of the N keys is searched, a pointer to the search function (*method*), and another pointer to the function which must be used to generate the keys to look for in the dictionaries (*generator*), which is already implemented in *search.c* and *search.h*. Besides, it receives *ptime*, a pointer to a structure (*TIME*) that when exiting the function will contain:

a) The size of the dictionary (*N*).
b) The number of searched keys (*n_elems*).
c) The average execution time (*time*).
d) The average case (*avg_ob*).
e) The best case (*min ob*).
f) The worst case (*max_ob*).

This routine will measure the average clock time, worst, best and average cases of a searching algorithm when searching keys within a table. The array of keys and the table will be created within 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);*

This function automatizes the time measurement. It calls the previous routine with a dictionary size which comes from *num_min* to *num_max*, using increments of size *incr.* It will also store the results in a file using the routine *save_time_table(char* file, PTIME time, int N)* implemented in another assignment.

We will check our code with the file *exercise2.c*.

3. Tools and methodology

In this practice, we both worked and coded in MacOS environment with the help of Atom. Since in MacOS we can execute and run in the terminal make commands as if it were Linux, we had no problem to check the exercises, but when we finished all the exercises, we had to use Linux to check valgrind errors.

3.1 Section 1

We first implemented the functions for dictionary ADT, whose structure was given in search.h, so that we could later work with them *init_dictionary* and *free_dictionary* were very straightforward since these types of functions are always very similar.

The function *insert_dictionary* was a little bit longer but since we had a little help with the pseudocode of the assignment pdf, it was not much more difficult than other function. Then we continued with *massive_insertion_dictionary*, which was just a special case of inserting elements into the dictionary, so we just made a loop in which we called *insert_dictionary*.

We use the function search_dictionary as a shortcut to search in the dictionary the key with the search function passed as the parameter method.

Once we have implemented all the functions related with the dictionary ADT, it is time to start with the search functions. First we code *lin_search* and then *bin_search,* which were very easy because we had previously implemented them in other courses. Finally, we did *lin_auto_search* which we did not understand at first, but once we did, we coded it with no problem.

To check all the functions we ran exercise1.c changing the searching function in order to check the three of them and without forgetting that when we call bin_search, the array has to be sorted.

3.2 Section 2

In this section we implemented two different functions in the times.c file. These functions are very similar to the ones we implemented in the first practice, but instead of working with simple arrays and and sorting functions, we use the dictionary ADT and search functions. Since this time we already knew and understood the behavior of these routines, it took us shorter to finish this exercise.

Then we continued to check what we had just coded and when we ran exercise2.c, we found that the average case was always really close to the worst case, which was wrong. We found the error was when calculating the average number of BOs because we did not divide by the actual number of addends N*n_times but by n*_times and that is why we got a greater number than we should.

4. Source code

4.1 Section 1

```
PDICT init_dictionary (int size, char order){
  PDICT dic = NULL;

  if (size < 0 || order > 1 || order < 0) return NULL;

  dic = (PDICT)malloc(sizeof(DICT));N
  if (!dic) return NULL;

  dic->size = size;
  dic->n_data = 0;
  dic->order = order;
  dic->table = (int *)malloc(size*sizeof(int));
  if (!dic->table){
    free_dictionary(dic);
    return NULL;
  }

  return dic;
}


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

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

  free(pdict);
  return;
}
```

```c
int insert_dictionary(PDICT pdict, int key)
{
  int a, j, counter = 0;
  if (!pdict) return ERR;
  if (pdict->n_data == pdict->size) return ERR;


  if (pdict->order == NOT_SORTED){
    pdict->table[pdict->n_data] =  key;
    pdict->n_data++;
    return counter;
  }

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

  a = pdict->table[pdict->n_data - 1];
  j = pdict->n_data - 2;

  while (j >= 0 && pdict->table[j]>a){
    counter ++;
    pdict->table[j+1] = pdict->table[j];
    j--;
  }
  if(j >= 0) counter++;

  pdict->table[j+1] = a;
  return counter;
}


int massive_insertion_dictionary (PDICT pdict,int *keys, int n_keys){
  int i, res, counter = 0;
  if (!pdict || !keys || n_keys < 0) return ERR;

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

  return counter;
}
```

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

  res = method (pdict->table, 0, pdict->n_data - 1, key, ppos);
  if (res == ERR) return ERR;

  return res;
}
```

Search algorithms

```
int bin_search(int *table,int F,int L,int key, int *ppos){
  int m, counter = 0;
  if (!table || F > L || F < 0 || !ppos) return ERR;

  while (F <= L){
    m = (F+L) / 2;

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

```c
int lin_search(int *table,int F,int L,int key, int *ppos){
  int i, counter = 0;
  if (!table || F > L || F < 0 || !ppos) return ERR;

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


int lin_auto_search(int *table,int F,int L,int key, int *ppos){
  int i, counter = 0;
  if (!table || F > L || F < 0 || !ppos) return ERR;

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

4.2 Section 2

```
short average_search_time(pfunc_search method, pfunc_key_generator generator,
                          char order, int N, int n_times, PTIME ptime){
  clock_t start, end;
  PDICT d = NULL;
  int *perms = NULL, *keys = NULL, i, res, pos, min = 0, max = 0;
  double sum = 0;

  if (!method || !generator || !ptime || n_times<=0 || N <= 0 || order > 1 || order < 0 ) return ERR;

  ptime->N = N;
  ptime->n_elems = n_times;

  d = init_dictionary (N, order);
  if(!d)return ERR;

  perms = generate_perm(N);
  if(!perms){
    free_dictionary(d);
    return ERR;
  }

  if (massive_insertion_dictionary (d, perms, N) == ERR) {
    free_dictionary(d);
    free(perms);
    return ERR;
  }

  keys = (int*)malloc(n_times * N * sizeof(int));
  if (!keys){
    free_dictionary(d);
    free(perms);
    return ERR;
  }

  generator(keys, N*n_times, N);

  start = clock();
```

```
        for (i = 0; i < n_times * N; i++){
          res = search_dictionary(d, keys[i], &pos, method);
          if (res == ERR){
            free_dictionary(d);
            free(perms);
            free(keys);
            return ERR;
          }
          if (min == 0) min = res;
          else if (res < min) min = res;
          if (res > max) max = res;
          sum += (double)res/(double)(N*n_times);
        }

        end = clock();

        ptime->time = ((double)(end - start) / (double)CLOCKS_PER_SEC) * 1000 / (double)(N*n_times);
        ptime->min_ob = min;
        ptime->max_ob = max;
        ptime->average_ob = sum;

        free_dictionary(d);
        free(keys);
        free(perms);

        return OK;
      }
```

```
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){
  int i_max, result, i;
  PTIME array_times = NULL;
  if (!method ||  !generator || !file || num_min>num_max || order > 1 || order < 0 || num_min<0 || incr<=0 ||n_times<=0) return ERR

  i_max = (num_max - num_min)/incr + 1;
  array_times = (PTIME)malloc(i_max*sizeof(TIME));

  if(!array_times) return ERR;

  for (i = 0; i < i_max; i++){
    result = average_search_time(method, generator, order, num_min + incr*i, n_times, &array_times[i]);
    if (result == ERR){
      free(array_times);
      return ERR;
    }
  }

  if (store_time_table(file, array_times, i_max) == ERR){
    free(array_times);
    return ERR;
  }

  free (array_times);
  return OK;
}
```

5. Results, plots

Results obtained for the different exercises, and their plots (if any).

5.1 Section 1

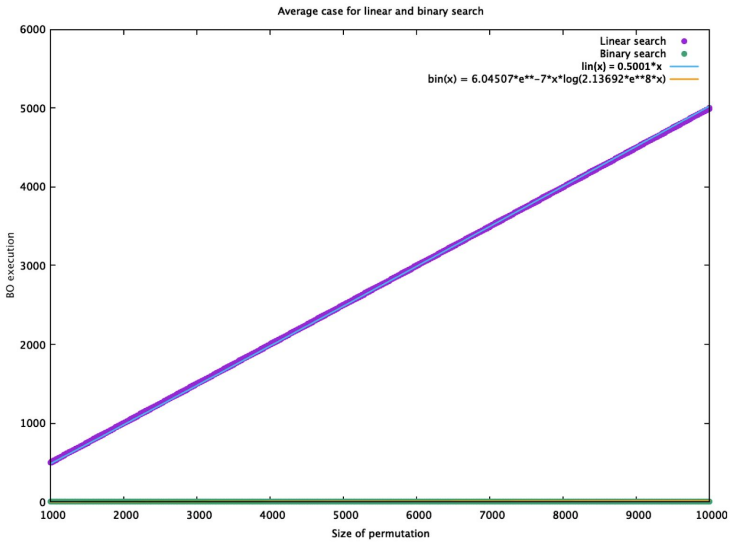
We check the search functions running exercise1.c changing the searching method in order to check the three different search routines and passing a sorted array when we call binary search.

In all of them we run ./exercise1 -size 10 -key 6, and obtain:

- lin_search: Key 6 found in position 8 in 9 basic op.
- bin_search: Key 6 found in position 5 in 3 basic op.
- lin_auto_search: Key 6 found in position 4 in 5 basic op.

5.2 Section 2

- Plot comparing the average number of BOs of linear and binary search approaches,



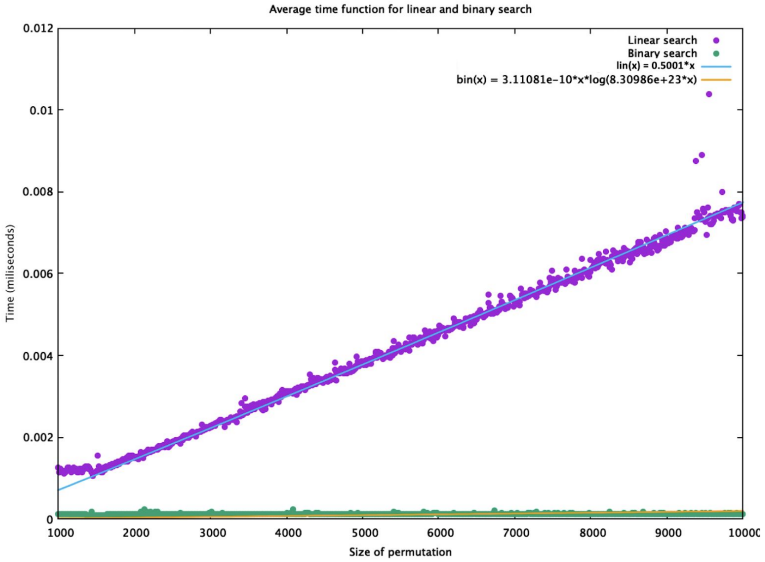
comments to the plot.

*plot "lin_search_1.log" u 1:3 with points pt 7 ps 0.7 title "Linear search", "bin_search_1.log" u 1:3 with points pt 7 ps 0.7 title "Binary search", a(x) title "lin(x) = 0.00781683*x*log(8.44164*e**23*x)" lw 1.5, w(x) title "bin(x) = 6.04507*e**-7*x*log(2.13692*e**8*x)" lw 1.5*

As we can see on the plot, linear search worse in the average case than binary search , since it grows much faster. In our plot binary search looks like an horizontal line at BO = 0 because there is a great difference between its graph and the one for linear search. For example, the

number of basic operation for a 7000 size-array is 6944 for binary search and 13 for linear search, which is a great difference.

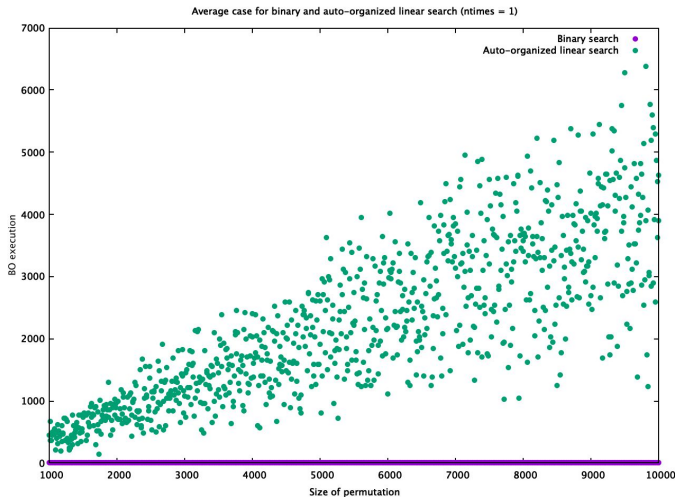
- Plot comparing the average clock time for the linear and binary search approaches, comments to the plot.



*plot "lin_search_1.log" u 1:2 with points pt 7 ps 0.7 title "Linear search", "bin_search_1.log" u 1:2 with points pt 7 ps 0.7 title "Binary search", g(x) title "lin(x) = 2.73165e-08*x*log(2.06106e+08 *x)" lw 1.5, t(x) title "bin(x) = 3.11081e-10*x*log(8.30986e+23*x)" lw 1.5*

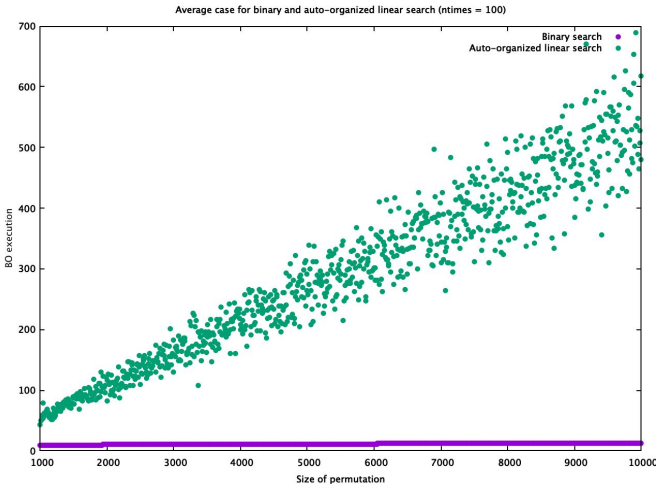
We can appreciate this plot resembles a lot the previous one. It is because, as with the average case for these algorithms, the average time is O(log n) for binary search and O(n/2) for linear search.

- 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.



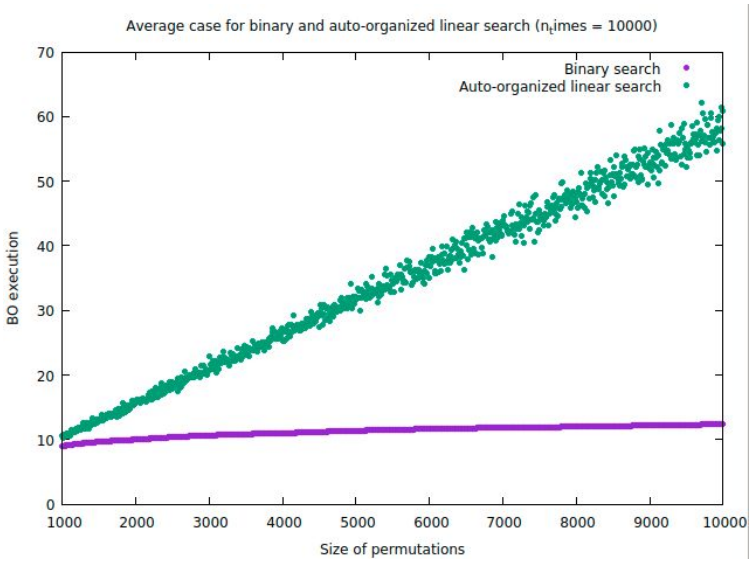
plot "bin_search_1.log" u 1:3 with points pt 7 ps 0.7 title "Binary search", "linauto_search_1.log" u 1:3 with points pt 7 ps 0.7 title "Auto-organized linear search"

It can be observed that in this plot the average number of BOs for linear auto-organized search are scattered whereas the ones for binary search are quite consistent. That is because since the average case for binary search is O(log n), having array sizes between 1000 and 10000, the results will vary within a range of 4 natural numbers.



plot "bin_search_100.log" u 1:3 with points pt 7 ps 0.7 title "Binary search", "lin_auto_search_100.log" u 1:3 with points pt 7 ps 0.7 title "Auto-organized linear search"

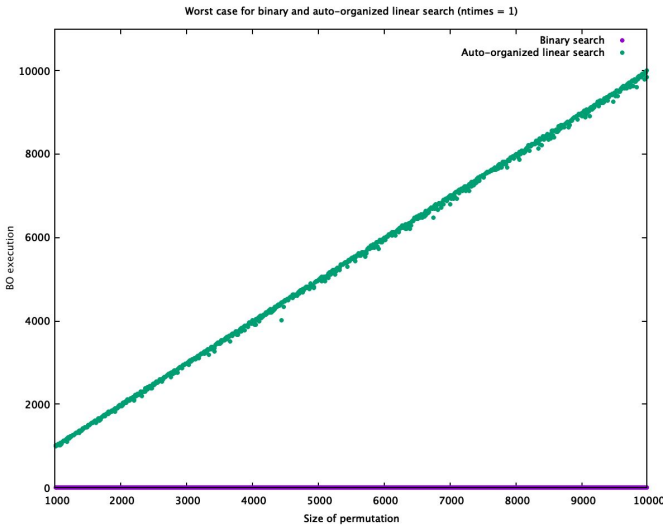
We can see that binary search average number of BOs in this case if quite similar to n_times = 1 and for lin_auto_search, they seem to be closer to each other, like if the points were approaching a line because the most times a key is searched using this algorithms, the fastest it will search.



plot "bin_search_10000.log" u 1:3 with points pt 7 ps 0.7 title "Binary search", "lin_auto_search_10000.log" u 1:3 with points pt 7 ps 0.7 title "Auto-organized linear search"

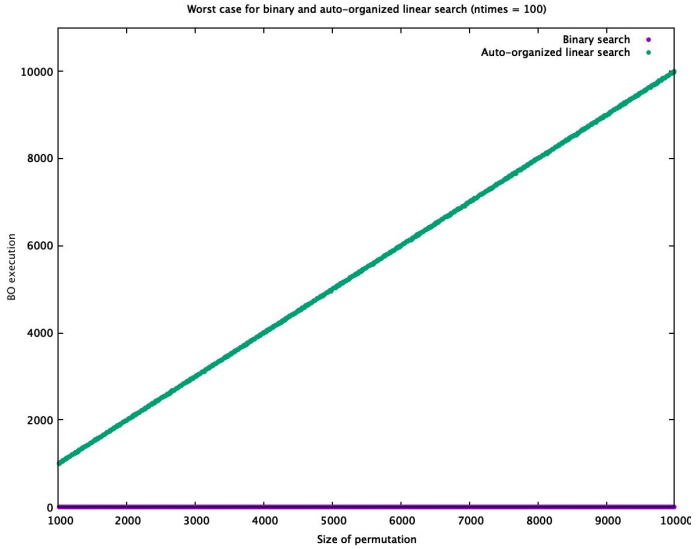
We observe that in this plot the binary search is the same as before but the average for lin_auto_search the BOs are much lower than before because the key is searched many more times so it is placed earlier in the array.

- 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.



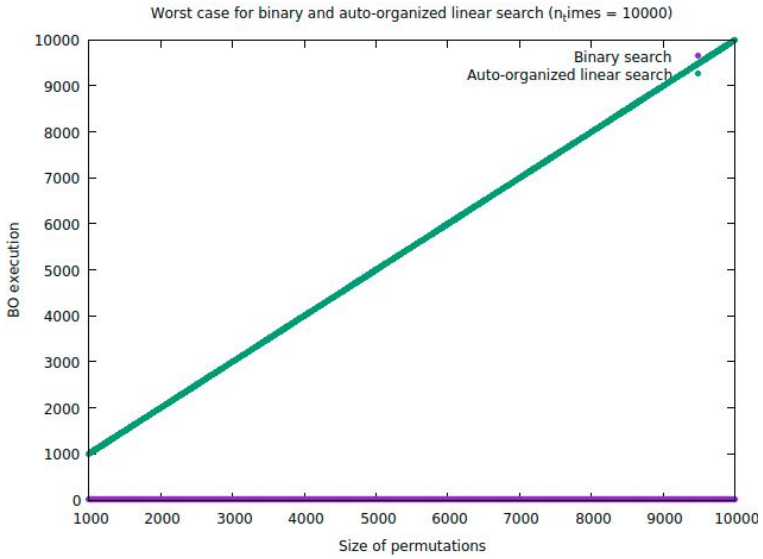
plot "bin_search_1.log" u 1:4 with points pt 7 ps 0.5 title "Binary search", "linauto_search_1.log" u 1:4 with points pt 7 ps 0.5 title "Auto-organized linear search"

In linear auto organized the worst case is N so it is obvious that the maximum for each size is N or at least very close to it, and that is why the green plot is almost a straight line. In binary search it happens as before, since it can only vary between 10 and 14 (log 1000 and log 10000, respectively), is is a "straight horizontal" line.



lot "bin_search_100.log" u 1:4 with points pt 7 ps 0.5 title "Binary search", "lin_auto_search_100.log" u 1:4 with points pt 7 ps 0.5 title "Auto-organized linear search"

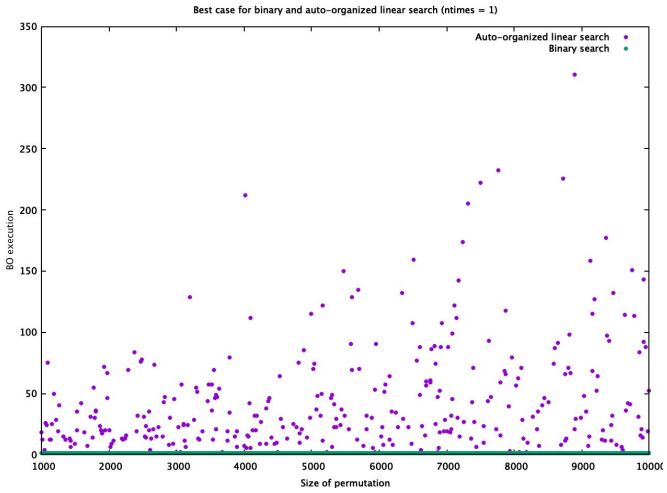
The plot for linear auto-organized search resembles this time even more a line like y = x because the more times a key is searched in an array, the more probable it is to reach the worst case. For binary search it is the same as before.



plot "bin_search_10000.log" u 1:4 with points pt 7 ps 0.7 title "Binary search", "lin_auto_search_10000.log" u 1:4 with points pt 7 ps 0.7 title "Auto-organized linear search"

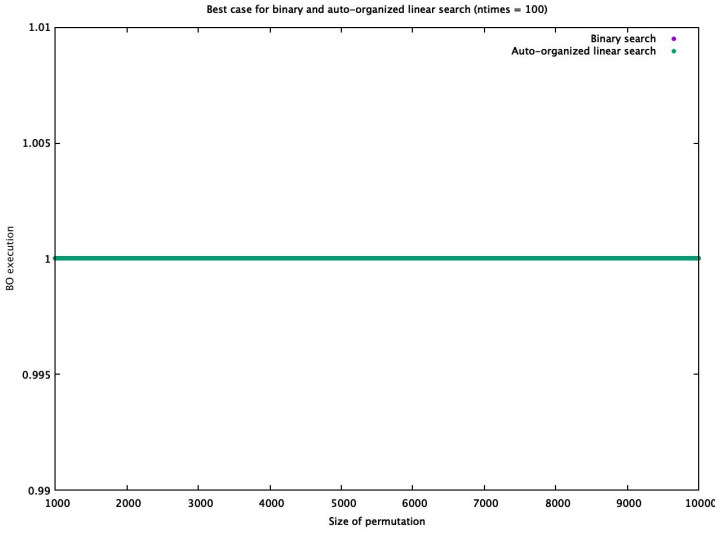
These time the auto-organized linear search is practically a straight line y = x, since the worst case of linear search is N. This is because the more times a key is searched, the more accurate the worst case will be.

- 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.



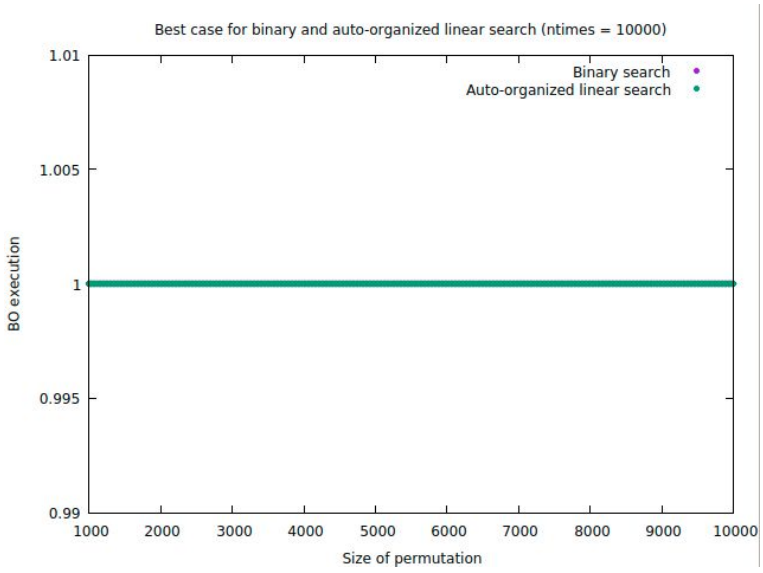
plot "lin_auto_search_1.log" u 1:5 with points pt 7 ps 0.5 title "Auto-organized linear search", "bin_search_1.log" u 1:5 with points pt 7 ps 0.5 title "Binary search"

In binary search the best case is 1 so the plot is a straight line because it finds the key in the middle of the table. It could have happened that for some arrays it was not 1 but it just did not occur. For lin_auto_search it happened as before, since n_times = 1 it is very difficult that is appears always in the first position, although it sometimes happens.



plot "bin_search_100.log" u 1:5 with points pt 7 ps 0.5 title "Binary search", "lin_auto_search_100.log" u 1:5 with points pt 7 ps 0.5 title "Auto-organized linear search"

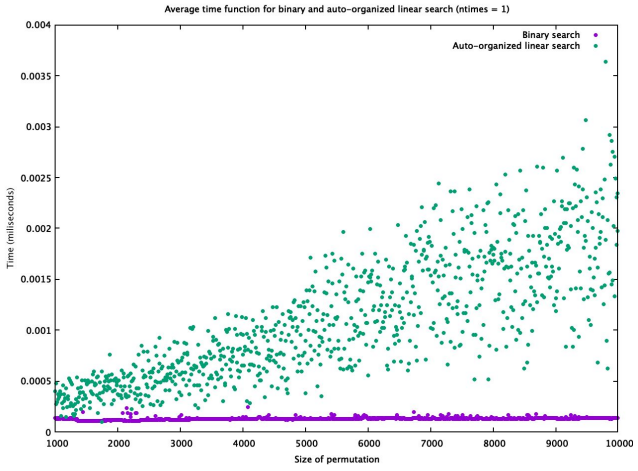
In this graph we can only see a green plot although 2 are being plotted. This is because since all the points take value 1 for BOs, then the two plots superimpose.



plot "bin_search_10000.log" u 1:5 with points pt 7 ps 0.7 title "Binary search", "lin_auto_search_10000.log" u 1:5 with points pt 7 ps 0.7 title "Auto-organized linear search"

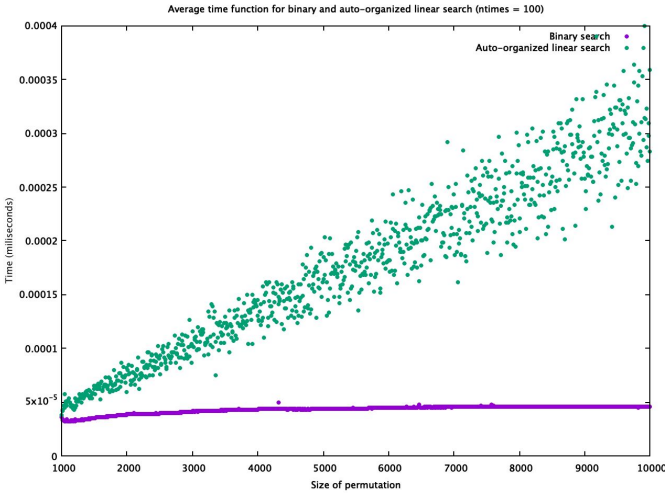
In this graph we get the same result as in the previous one. We can only see a green plot, since the best case for both algorithms is 1 (also for the auto-organized linear search, since n_times is really high), then the two plots superimpose.

- 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.



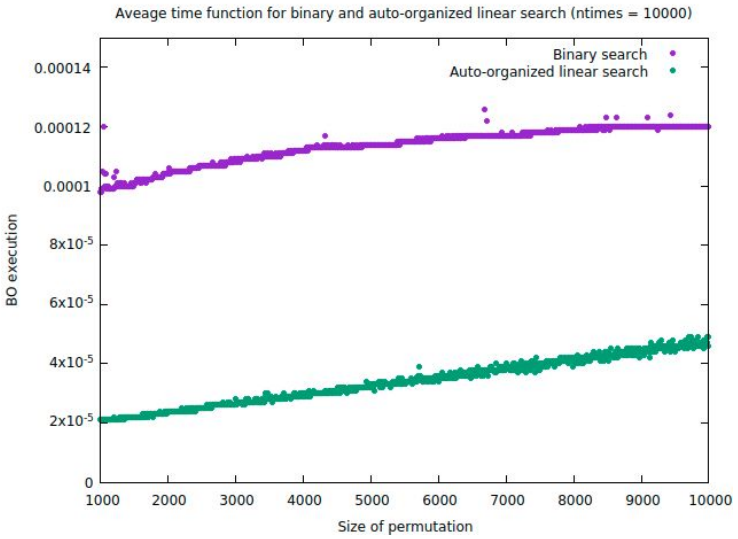
plot "bin_search_1.log" u 1:2 with points pt 7 ps 0.5 title "Binary search", "linauto_search_1.log" u 1:2 with points pt 7 ps 0.5 title "Auto-organized linear search"

We can appreciate this plot resembles a lot the average case plot, since all the points for linear auto-organized search are scattered whereas the ones for binary search are quite consistent. It is because, as with the average case for these algorithms, the average time is O(log n) for binary search, but since this plot is for n_times = 1, the table of linear auto-organized search is not sorted yet.



plot "bin_search_100.log" u 1:2 with points pt 7 ps 0.5 title "Binary search", "lin_auto_search_100.log" u 1:2 with points pt 7 ps 0.5 title "Auto-organized linear search"

In this plot, the result for binary search is the same as in the previous one, but the plots of auto-organized linear search are less scattered, since this plot is for n_times = 100, so the array of the auto-organized linear search is more sorted.



plot "bin_search_10000.log" u 1:2 with points pt 7 ps 0.7 title "Binary search", "lin_auto_search_10000.log" u 1:2 with points pt 7 ps 0.7 title "Auto-organized linear search"

Now the auto-organized linear search curve is lower than the binary search. This is because since this plot is for n_times = 10000, the array of auto-organized linear search is completely sorted, so it will be really quick to find the keys, since they are sorted by probability.

5. Response to the theoretical questions.

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

- lin_search: table[i] == key
- bin_search: if (table[m] > key){...}

  else if (table[m] < key){...}
- lin_auto_search: table[i] == key

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.

Linear search: W(N) = N

B(N) = 1

Binary search: W(N) = log(N) + O(1)

B(N) = 1

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?

In lin auto search, when the key is found (table[i]), we perform the following operation: *swap(&table[i], &table[i-1])* → we swap the searched element with the previous one.

As long as the number of searches increases, the most searched elements will we positioned at the beginning of the table, since each time we search a key, we will bring it forward in the table.

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.

At the beginning, when the array is not yet sorted , the average case of lin_auto_search can be calculated as follows:

$C_N = \sum_{i=1}^{N} \frac{log(i)}{i}$ and $S_N = \sum_{i=1}^{N} \frac{i * log(i)}{i} = \sum_{i=1}^{N} log(i)$

If we calculate these summations using integrals as learnt in theory class, we obtain that $A_{lin\_auto\_search}(N) \sim \frac{2N}{log(N)}$

As more keys are being searched and the arrays becomes sorted, the probability of searching the 80% of the queries ask for the 20% of the keys following the rule of the 80/20. Then when calculating the average case we get:

$A_{lin\_auto\_search}(N) = 0.8 * 0.2N / 2 + 0.2 * (N - 0.2N)/2 \approx 0.16N$

Following Zipf law as commented in the assignment, we can conclude

$$A_{lin\_auto\_search}(N) = O(1)$$

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

This algorithm works because in each iteration if the middle element is the searched key, it returns and the key is found. In any other case, it would divide the array in two, reducing its size to its half. Since the array is sorted, any element is greater than all the other elements on its left and, consequently, any element is smaller than all the others to its right.

If the value of the key is greater than the element in the middle, the new array would be the one on the right and when the key is smaller the new array would be the half on the left hand side.

This process is repeated until the key is found in the middle of the the array or until the last element is the same as the first element, which means the key is not in the array.

6. Conclusions.

In this practice we have learnt and fully understood the theory studied in this course of analysis of algorithms regarding dictionaries and searching algorithms. But not only that, we believe that we have improve our skills while implementing certain functions or algorithms as well as for solving problems that may come up while coding. Some of this problems can be syntax errors, problems with pointers and double pointers but mainly with valgrind. Besides, we believe we have improved our skills when it comes to check errors within functions, since that was our major mistake on the previous assignments.

We wanted to highlight the fluency we have acquire when it comes to understanding and programming these algorithms, since we now understand their different methodologies and uses. This assignment puts into practice the complexity of the algorithms learnt during theory lessons, because we were able to see how the best, worst and average case of an algorithm work on a real environment. We confirmed that experimental values actually meet with the values we can theoretically calculate. This is represented in the graphics we created to illustrate the results.