Analysis of Algorithms 2020/2021

Practice 3

Pablo Cuesta Sierra and Álvaro Zamanillo Sáez. Group 1251.

Code	Plots	Documentation	Total

1. Introduction.

In this practice, we are going to implement the Dictionary ADT in order to measure the average, best, and worst case for different search algorithms (binary search, linear search and linear auto search).

2. Objectives

2.1 Section 1

In this section we implement the functions that create, free, insert and search in the dictionary structure. For the search function, different functions can be used.

2.2 Section 2

In this section we implement the functions generate_search_times and average_search_time, which are used later in order to measure systematically the efficiency of the different search functions implemented in the previous section.

3. Tools and methodology

For this practice we have used again GitHub to share the code and keep track of the versions; for checking errors in the code regarding memory, valgrind; for compiling, gcc; and finally, for plotting the results obtained, GNU Plot.

3.1 Section 1

The functions in this section are very straightforward, we had to be careful, though, to make sure we counted the number of basic operations correctly and that the return parameters were as requested.

3.2 Section 2

These functions were very similar to those in the practice 1, where we coded generate_sorting_times and average_sorting_time, so these functions were almost the same, we only had to make use of the dictionaries in this case. And we also reused the function save_time_table implemented in practice 1.

4. Source code

Here, include the source code only **for the routines you have developed** for each exercise.

4.1 Section 1

```c
#define MIN(a, b) ((a)<(b))?(a):(b)

PDICT init_dictionary (int size, char order)
{
    PDICT pd=NULL;
    /*arguments check*/
    if((order!=SORTED && order!=NOT_SORTED)||(size<=0)){
        return NULL;
    }
    /*memory allocation of the dictionary*/
    if(!(pd=(PDICT)calloc(1, sizeof(DICT)))){
        return NULL;
    }
    /*memory allocation of the table that contains the data*/
    if(!(pd->table=(int*)calloc(size, sizeof(int)))){
        free(pd);
        return NULL;
    }
    /*initialize the data of the structure*/
    pd->size = size ;
    pd->n_data = 0 ;
    pd->order = order ;

    return pd;
}

void free_dictionary(PDICT pdict)
{
    if(pdict!=NULL){
        free(pdict->table);
        free(pdict);
    }
    return ;
}

/*Auxiliary function to resize the table of a dictionary to the new_size */
int resize_dictionary(PDICT pdict, int new_size){
    int *aux = NULL;

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

    if(new_size<=(pdict->size)){
        return OK; /*no need to realloc*/
    }

    if((aux = (int *)realloc(pdict->table, new_size))==NULL){
```

```
        /*there is an error, no memory left, but the
        data is not lost because we used an auxiliary pointer*/
        return ERR;
    }else{
        pdict->size = new_size ;
        pdict->table = aux;
        return OK;
    }
}

int insert_dictionary(PDICT pdict, int key){
    int j = 0, BO = 0;

    if(pdict==NULL)
        return ERR;

    /*resize if needed*/
    if((pdict->size)<(pdict->n_data + 1)){
        if(resize_dictionary(pdict, (int)(2*(pdict->size)))==ERR){
            /*no data was lost, but there is not memory enough to insert a new key*/
            return ERR;
        }
    }
    if(pdict->order==NOT_SORTED){
        /*insert at the end of the table*/
        pdict->table[pdict->n_data] = key;
        (pdict->n_data)++;
    }else{
        j = pdict->n_data;
        /*add it at the end of the table */
        pdict->table[j] = key;
        (pdict->n_data)++;
        j--;
        /*insert the key in its place: */
        while(j>=0 && pdict->table[j]>key){
            pdict->table[j+1] = pdict->table[j];
            j--;
        }
        BO=MIN((pdict->n_data)-j, (pdict->n_data)-1);
        /**this is the number of KC comparisons made:MIN
         * used in case j==0, because there are only
         * (pdict->n_data) elements, so the key can be
         * compared only to (pdict->n_data)-1
         **/
        pdict->table[j+1] = key;
    }
    return BO;
}
```

```c
int massive_insertion_dictionary (PDICT pdict,int *keys, int n_keys){
    int i=0, ret=OK, BO=0;

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

    for(i=0; i<n_keys && ret!=ERR; i++){
        ret=insert_dictionary(pdict,keys[i]);
        BO+=ret;
    }

    if(ret==ERR)
        return ERR;

    return BO;
}

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

    if(!method)
        return ERR;

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


/* Search functions of the Dictionary ADT */
int bin_search(int *table,int F,int L,int key, int *ppos){
    int M=0, BO=0;

    (*ppos)=NOT_FOUND;

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

    /*non recursive*/

    while(F<=L){
        M=(F+L)/2;
        BO++;
        if(table[M]<key){
            F=M+1;
        }
        else if(table[M]>key){
            BO++;
            L=M-1;
        }
        else{
```

```
            BO++;
            (*ppos)=M;
            break;
        }
    }

    return BO;
}

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

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

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

    return bo;
}

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

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

    for (i = F; i <= L; i++){
        bo++;
        if (table[i] == key){
            if (i != F){
                table[i] = table[i - 1];
                table[i - 1] = key;
                *ppos = i - 1;
            }
            else{
                *ppos = i; /*We return the position where the element ends after the
searc*/
            }

            return bo;
        }
    }
```

```
    *ppos=NOT_FOUND;
    return bo;
}
```

4.2 Section 2

```
short generate_search_times(pfunc_search method, pfunc_key_generator generator, int o
rder, char* file, int num_min, int num_max, int incr, int n_times){

    int N, i, n_iter;
    TIME_AA *ptime = NULL;

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

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

    if (!(ptime = calloc(n_iter, sizeof(TIME_AA))))
        return ERR;

    for (i = 0, N = num_min; i < n_iter; N = num_min + (i)*incr){
        if (average_search_time(method, generator,order, N, n_times, ptime + i) == ER
R){
            free(ptime);
            return ERR;
        }

        printf("Num %d\n", N); /*To check that the program is running*/
        i++;
    }

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

    free(ptime);

    return OK;
}

short average_search_time(pfunc_search metodo, pfunc_key_generator generator, int ord
er, int N, int n_times, PTIME_AA ptime){
```

```c
int *keys,*perm,i,n_keys,ppos,min,max;
long c_aux=0,c_total=0;
PDICT dict=NULL;
clock_t begin, end;

if(!metodo||!generator||(order!=SORTED && order!=NOT_SORTED)||N<0|| !ptime){
    return ERR;
}
/*Allocate dictionary, permutations and fill the dictionary*/
if(!(dict=init_dictionary(N,(char)order)))
    return ERR;

if(!(perm=generate_perm(N))){
    free_dictionary(dict);
    return ERR;
}

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

/*Allocate keys array and fill it with key_generator function*/
n_keys=N*n_times;

if(!(keys=(int*)calloc(n_keys,sizeof(int)))){
    free_dictionary(dict);
    free(perm);
    return ERR;
}

generator(keys,n_keys,N);

begin = clock();
/*gettimeofday(&start, NULL);*/
for(i=0;i<n_keys;i++){
    c_aux=metodo(dict->table,0,N-1,keys[i],&ppos);

    if(!i){
        min = max = c_aux;
    }else{
        min = c_aux < min ? c_aux : min;
        max = c_aux > max ? c_aux : max;
    }
    c_total += c_aux;
}
end = clock();
```

```
    ptime->time = (double)(end - begin) / (double)(CLOCKS_PER_SEC*n_keys);
    ptime->average_ob = (double)c_total / (double)n_keys;
    ptime->max_ob = max;
    ptime->min_ob = min;
    ptime->N = N;
    ptime->n_elems = n_keys;

    /*Free memory*/
    free_dictionary(dict);
    free(perm);
    free(keys);

    return OK;
}
```

5. Results, plots

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

5.1 Section 1

This section was checked using the provided exercise1.c, where we could see that our dictionary worked correctly: the insertion and the search functions worked as expected.

5.2 Section 2

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

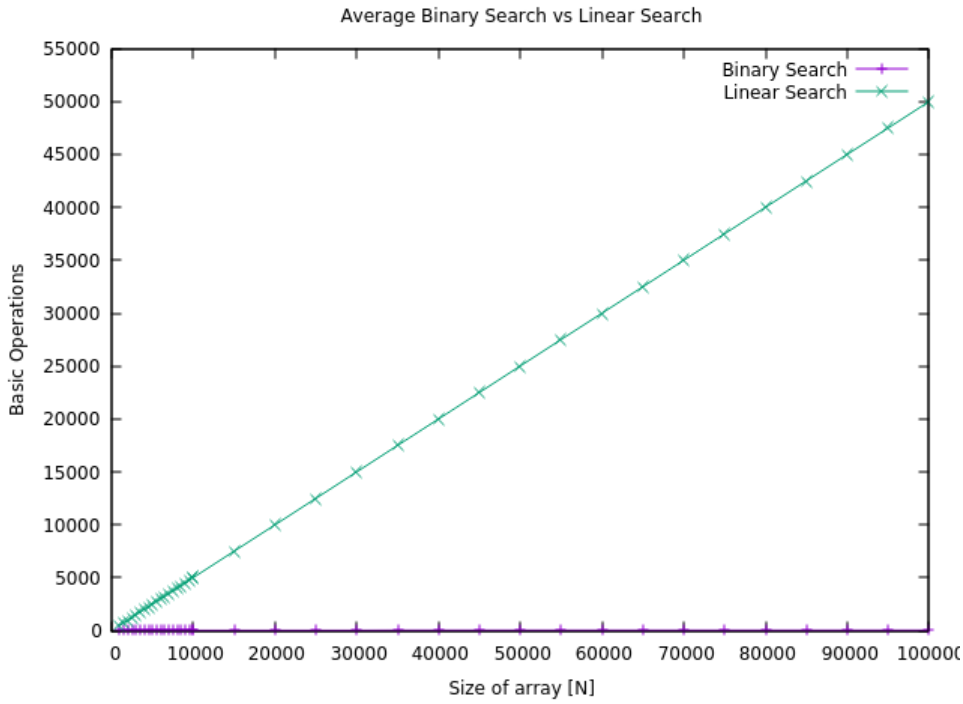

Figure 1

As we see in Figure 1, the linear search makes on average N/2 key comparisons. Binary search, on the other hand has an average case of Θ(log N), which can be seen on Figure 2, because in Figure 1, the size of the linear function is much bigger, and the binary search function cannot be appreciated:

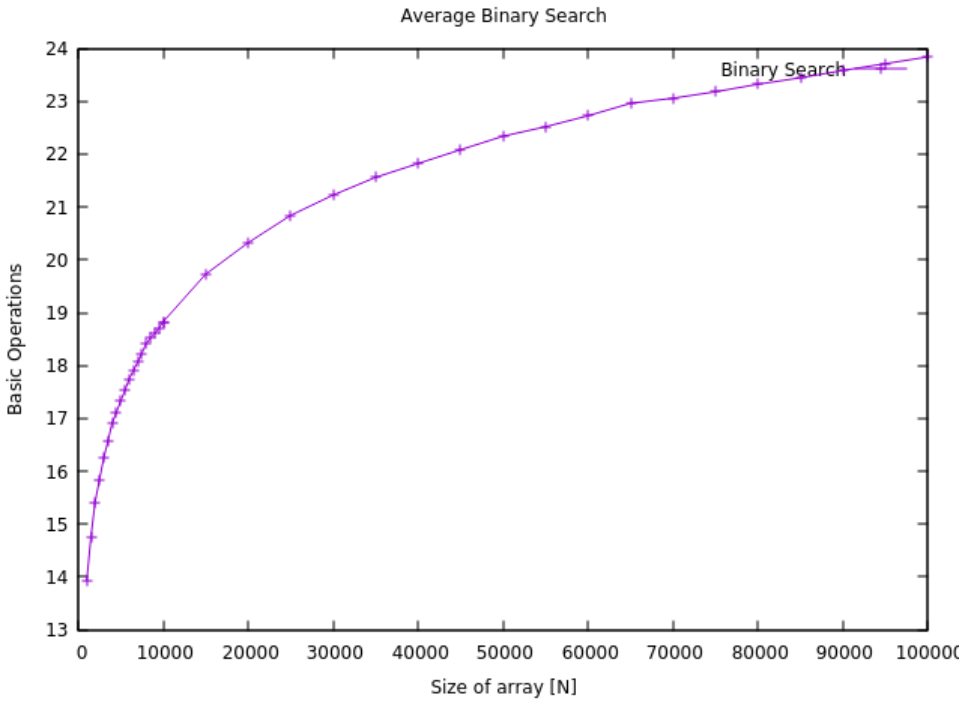

Figure 2

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

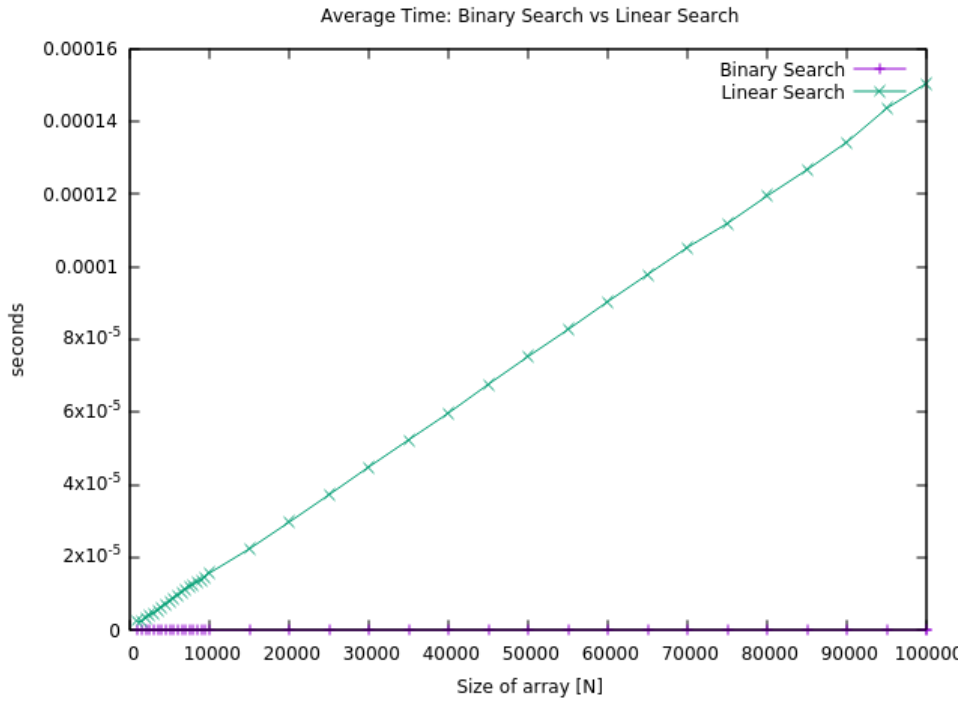

Figure 3

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.

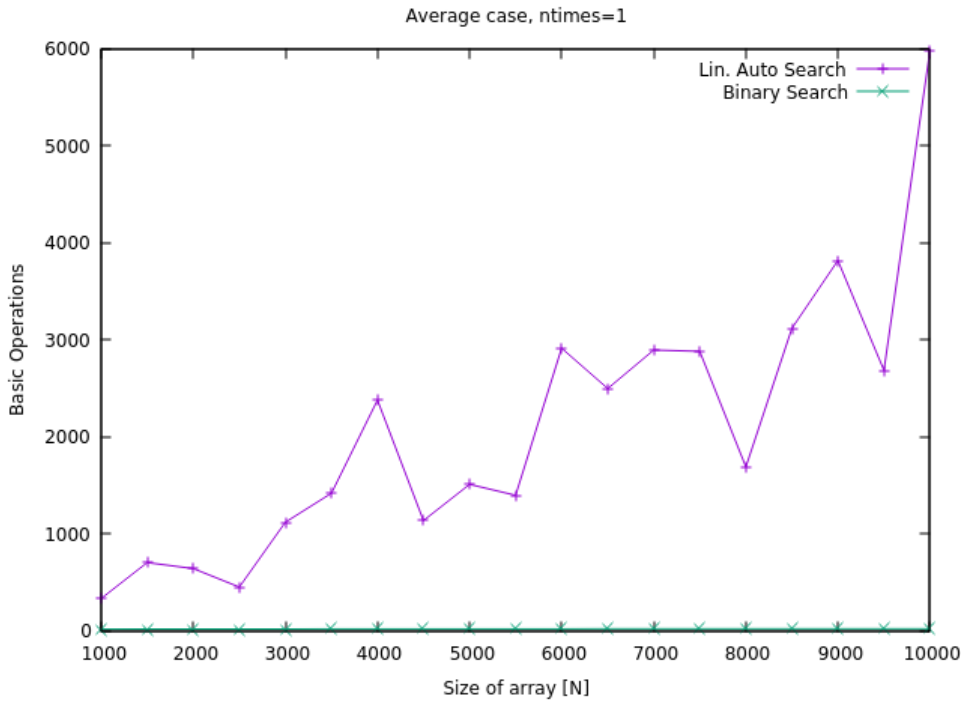

Figure 5

For n_times=1, binary search has a very low average case, just like in Figure 1 and Figure 2, it is a function that is O(logN), because even the worst case in binary search is O(logN), so how the keys are generated does not affect the average time very much.

However, for the lin_auto_search, as for each N, the permutation is different, and the more frequent keys can be lower or higher in the table. This means that even if they get closer to the beginning each time they are searched, if they are close to the end, for that particular N, there will be a very high BO count. That is why in Figure 5, we see some peaks in the table. The opposite happens if the frequent values happen to be closer to the beginning: the overall BO count will be lower. This was also expected, as it was suggested in the assignment pdf.

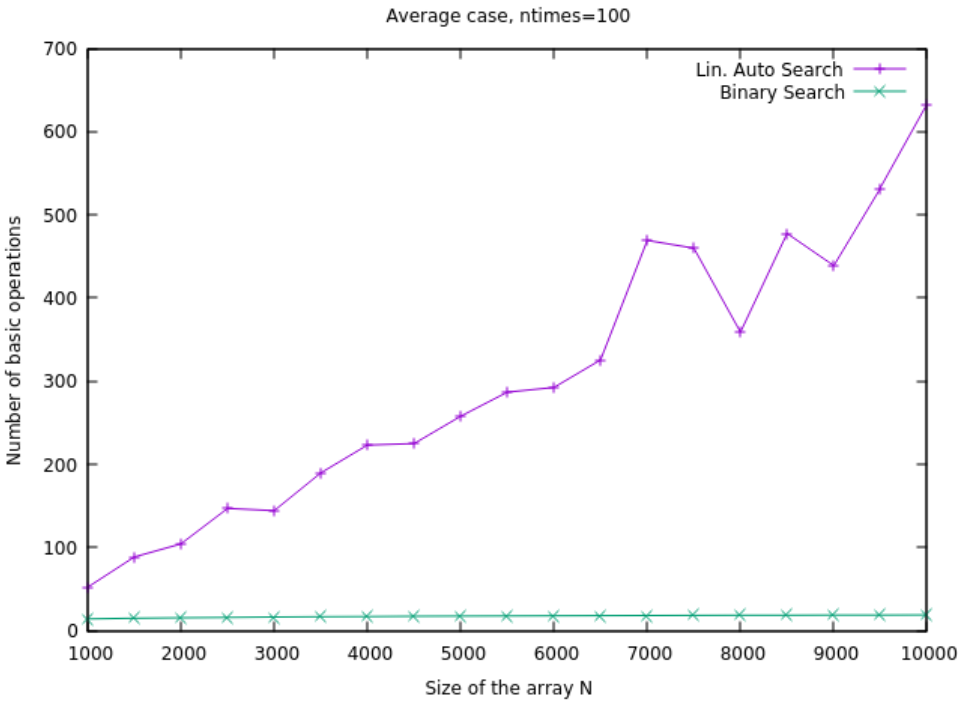

Figure 6

This time, for n_times=100, there is more flexibility for the table in the linear auto search algorithm to be modified in order to have the most searched keys at the beginning, so we can see that, at least for lower values, the function is more stable, and, it seems, O(N). For the higher values, though, there are still some peaks.

Binary search, as expected, does not vary much.

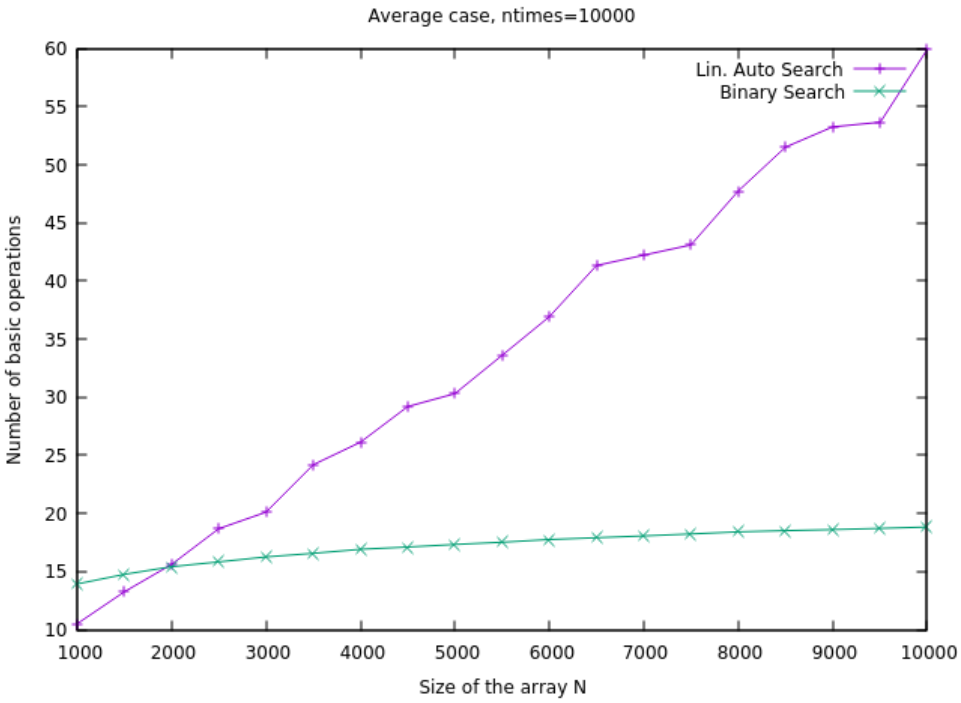

Figure 7

In Figure 7 we can see that the function of the average case of linear auto search gets a lot more stable. Also, we can see that all the values (of linear auto search) are a lot lower the higher n_times is. For N=10000, in Figure 5, the average BO count is 5971.62, in Figure 6, 632.26, and in Figure 7, 59.96. So, it decreases at a very high rate.

While binary search still is O(logN) and does not change with different n_times values.t

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.

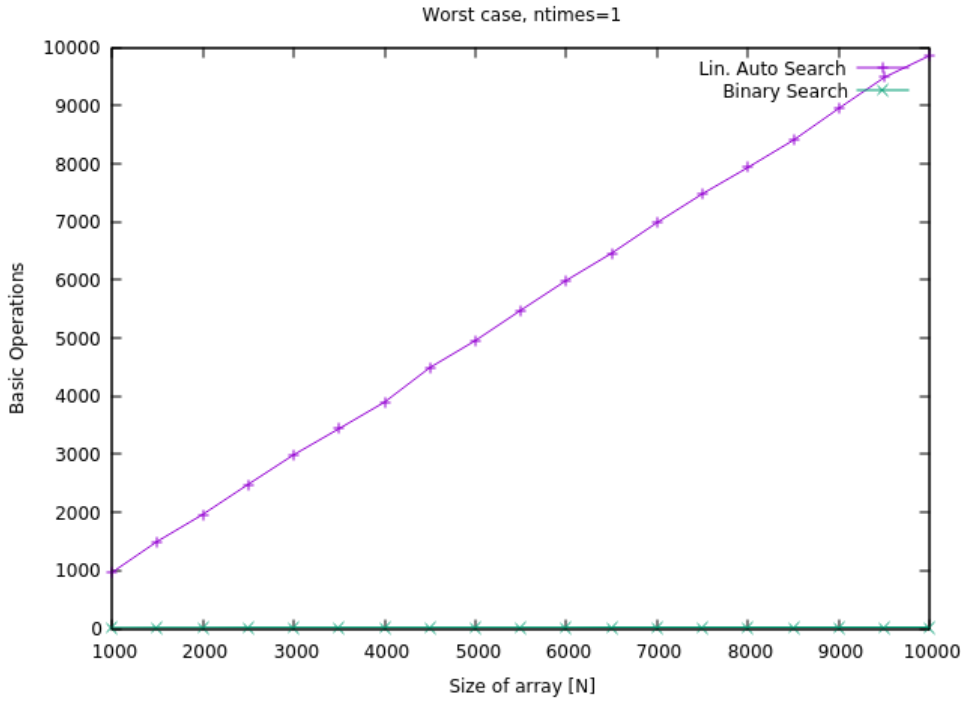

Figure 8

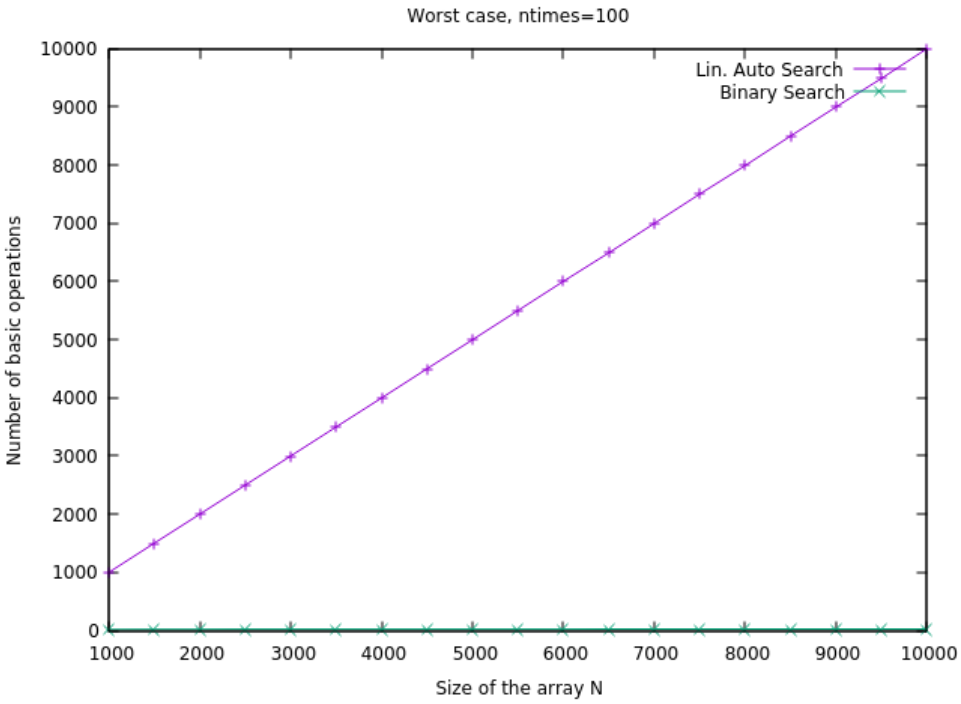

Figure 9

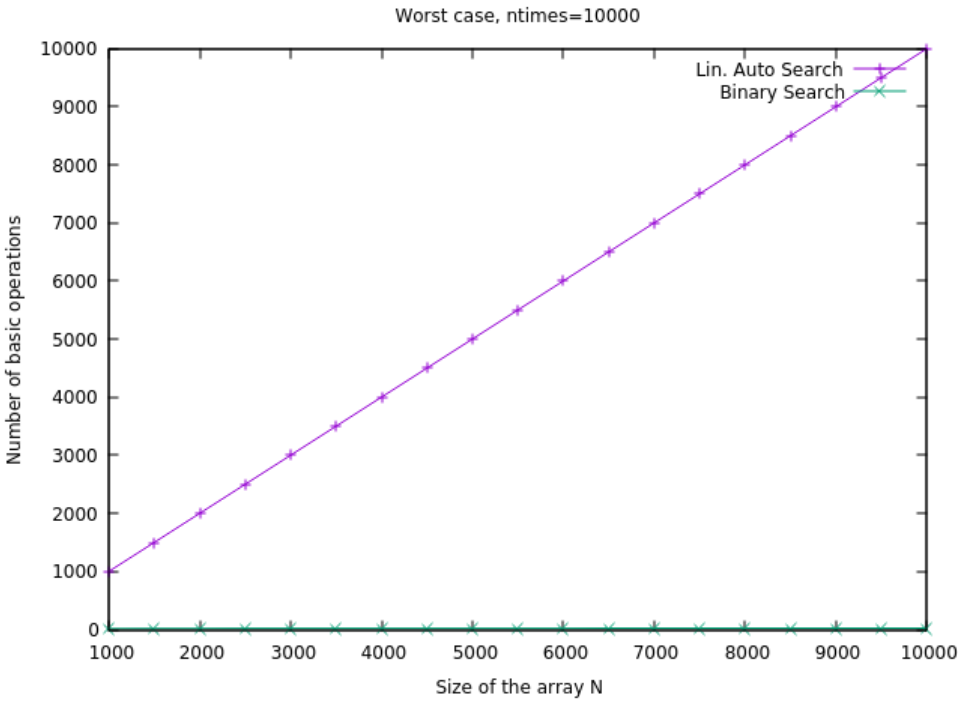

Figure 10

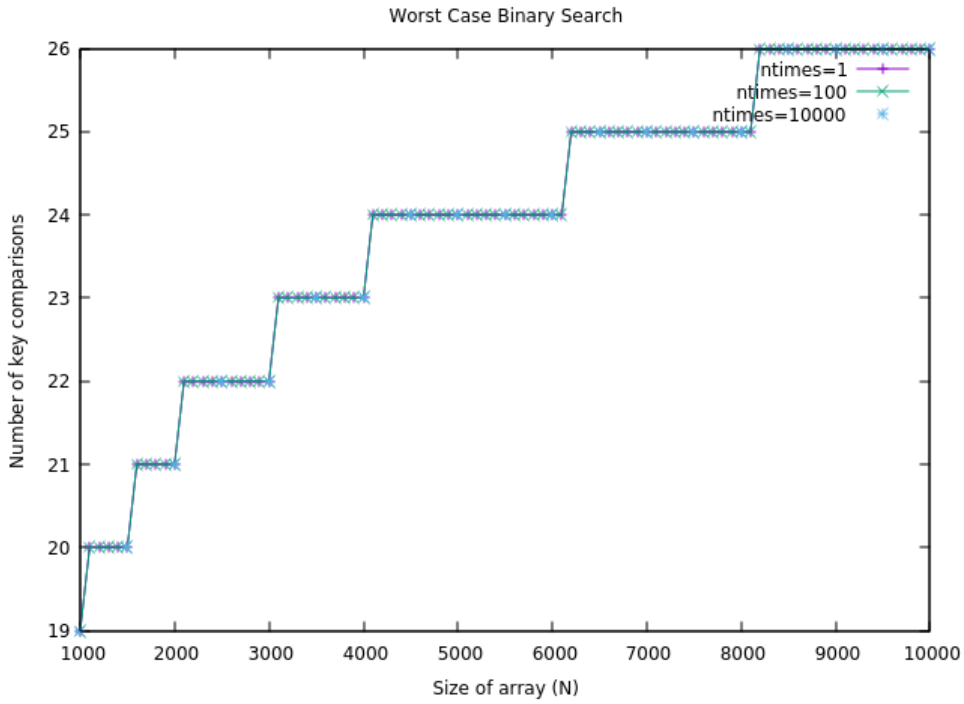

Figure 10.1

The worst case does not change significantly with different values for n_times, as we see in Figures 8, 9, and 10. This is normal, because even if not all possible keys have the same probability of appearing, it is very likely that at least one of the keys that appears is within, say, the last 100 elements of the array, which makes the BO count almost as high as it can get. Of course, the higher n_times is, the more likely it is that the maximum approaches N, but it is so close that we cannot appreciate the difference looking at the plots. For the binary search, just like with the average, this potential generation does not alter the fact that the worst case is always lower or equal to floor(logN), that is, O(logN).

In figure 10.1, we can see that (obviously) binary search does not depend on the number of times each key is searched as the array is not being modified. Furthermore, we can appreciate the tendency which is logarithmic (with discrete values). And the worst case (for array of size N) of the order of the height of the (balanced) Binary Search Tree of size N, which is floor(logN), that is why we see this plot (Figure 10.1).

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.

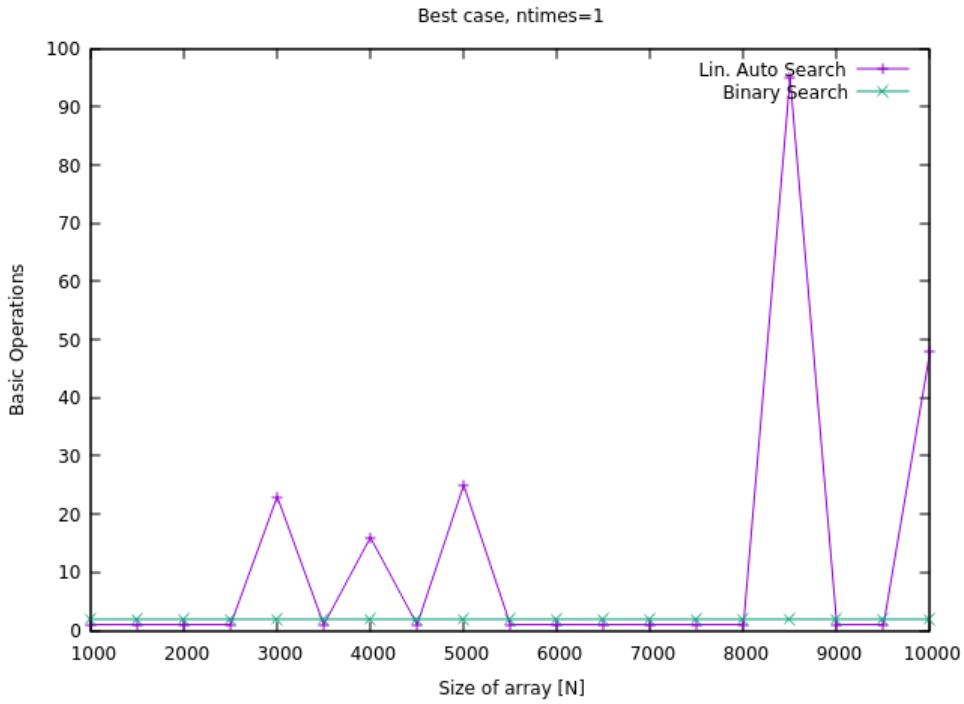

Figure 11

Here we can see that for binary search the minimum BO count is always reached (it corresponds to finding the searched key in the middle position of the array), and this count is 2, because with our code, we compare first whether it is smaller than the key, then if it is bigger (that's already two comparisons), and the default case (if it is neither smaller nor bigger) is that it is equal: so, the best case is 2 in this case.

In Linear Auto Search, as the generation of keys does not follow equiprobability, not in all cases the

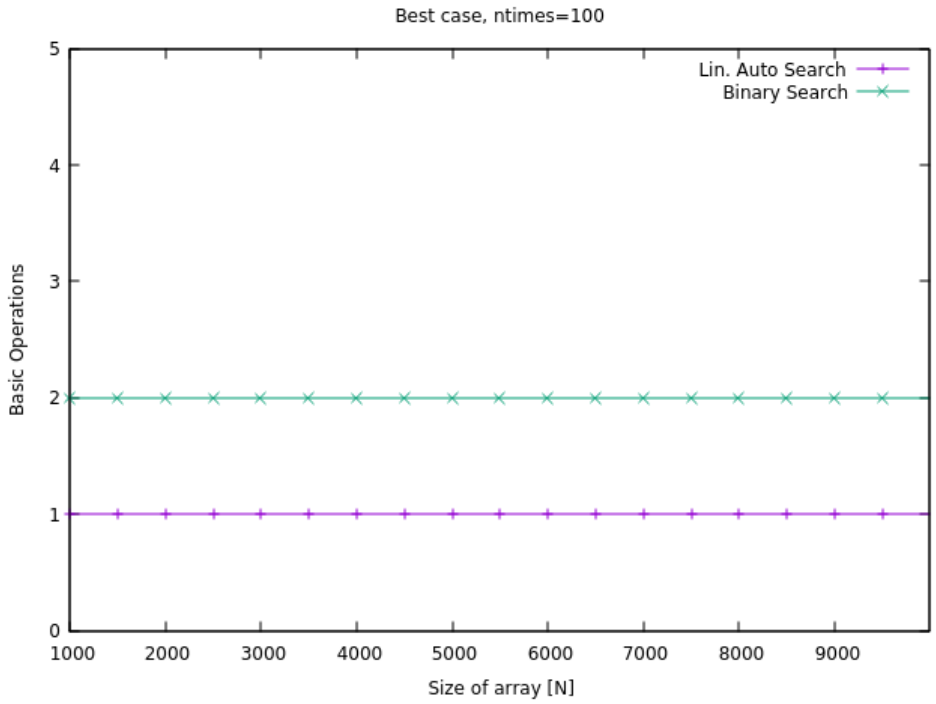

Figure 12

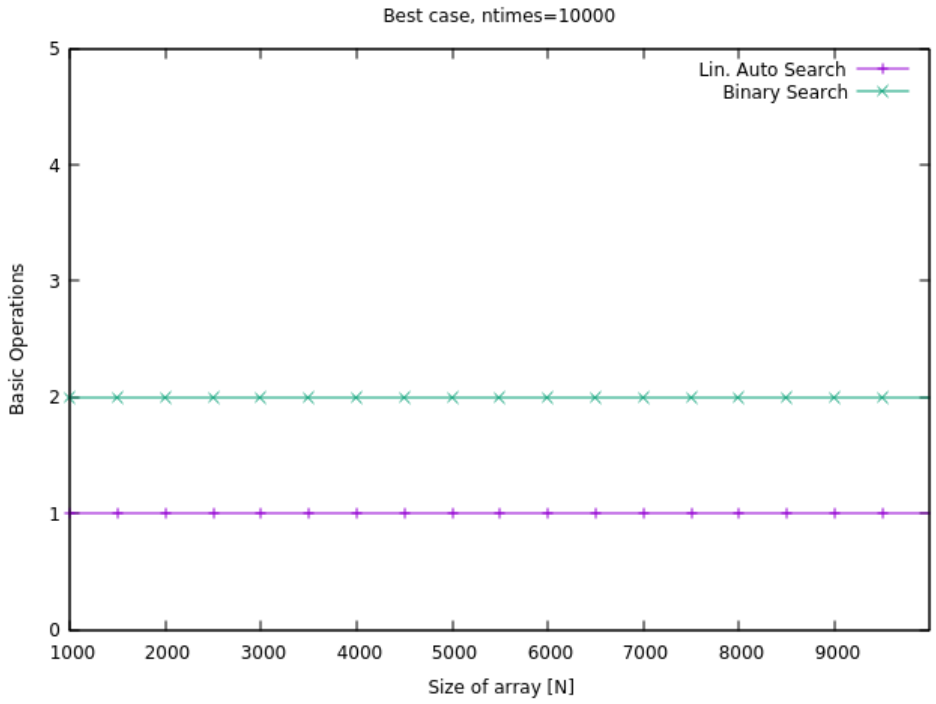

Figure 13

In Figures 12 and 13 (which have the same results), we can see that for n_times as big as 100 or 10000, the best case is reached for both algorithms. As has been explained in the comment of Figure 11, this means that the Best Case for Binary Search as we have coded it, is 2. And for Linear Auto Search, it is 1, that corresponds to finding the key in the first element of the array.

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.

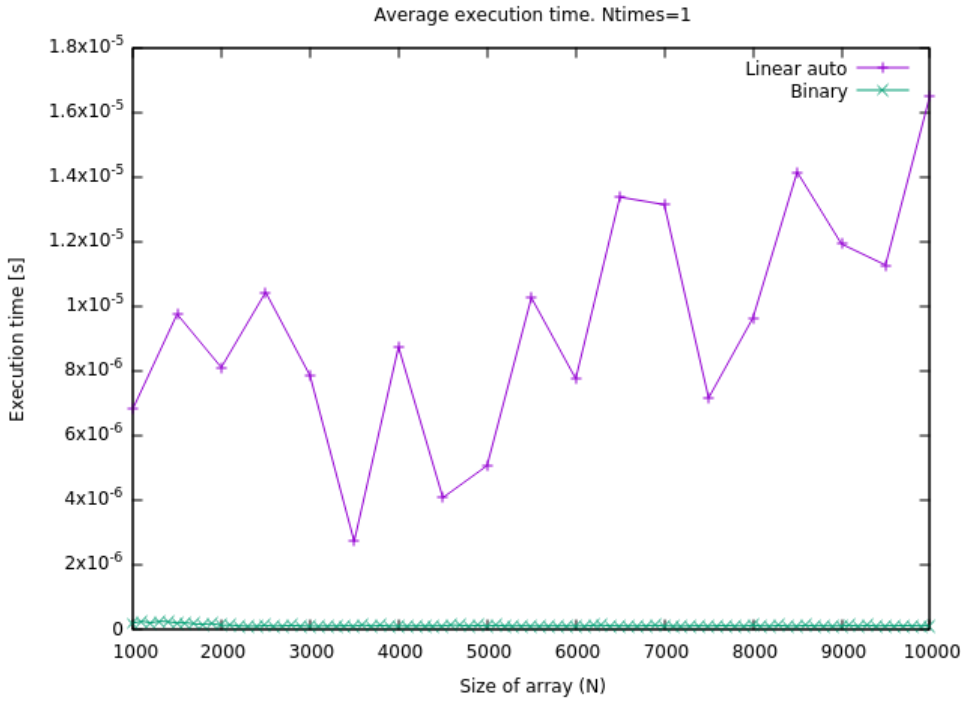

Figure 14

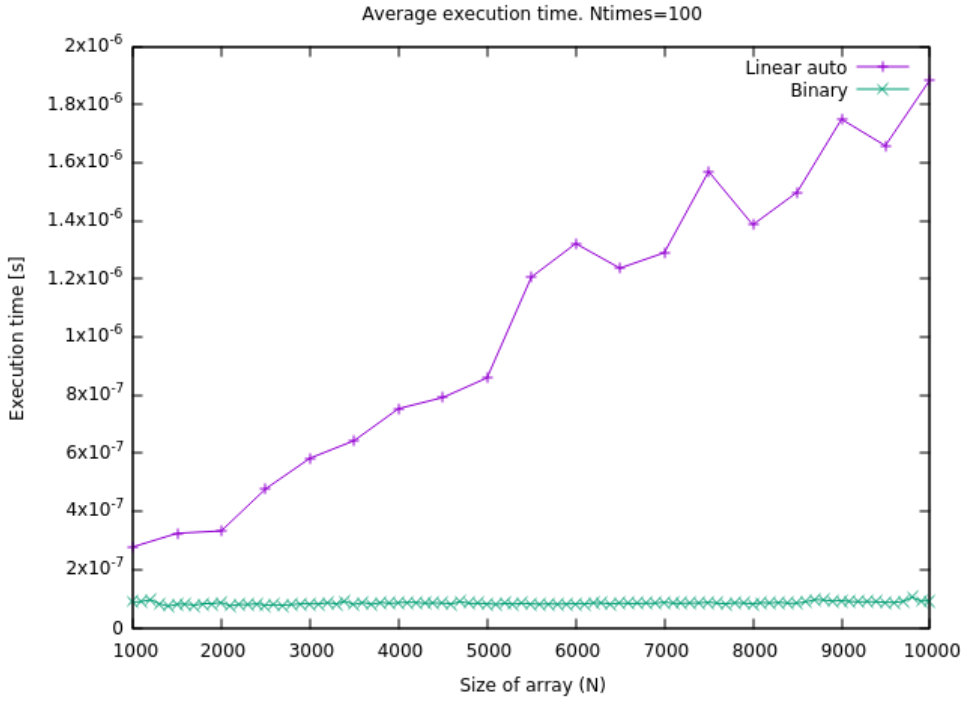

Figure 15

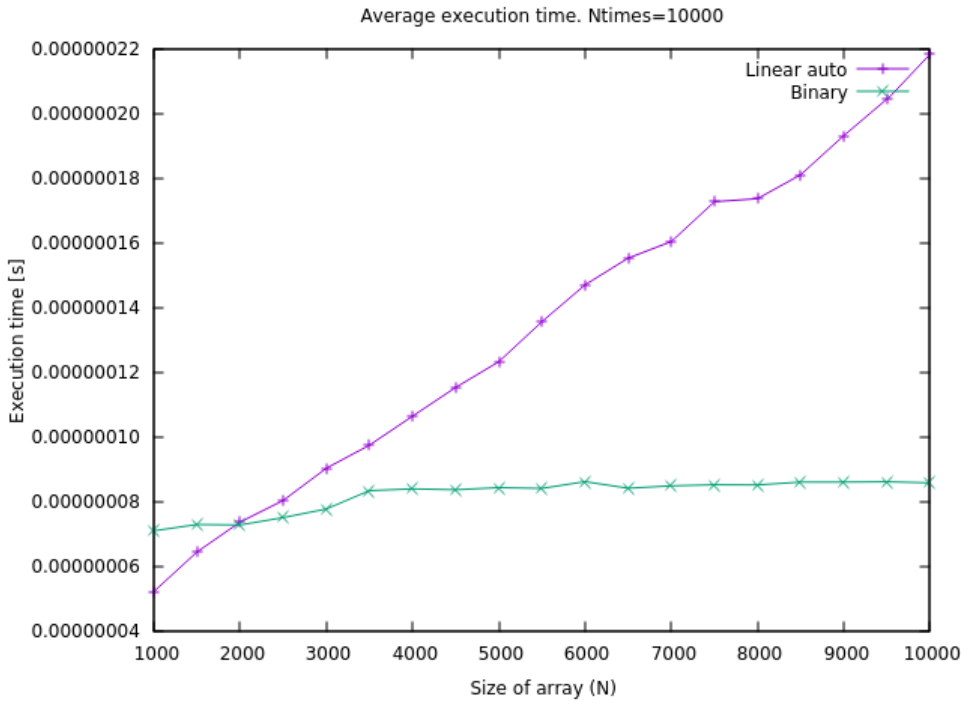

Figure 16

After comparing figures 14, 15 and 16, it is easy to note the difference between a uniform generation of keys and a potential generation when using the auto-linear search, as well as the effect of the number of times each key is searched.

When a key is searched frequently it gets closer to the beginning of the array, and therefore, the execution time needed to find it gets reduced significantly. In the case of figure 14, ntimes was 1 so the effect of the auto search was not as noticeable as in the others figures: the most common searched keys could not reach the first positions of the array because the number of times they were searched was not high enough. When compared with the binary search, we can see that the linear auto search is orders of magnitude slower than binary.

However, when the factor ntimes is increased, the execution time of the linear search gets closer to the binary search results (in terms of order of magnitude). In figure 15 we can see that linear search is still 10 times slower (for high values of N) whereas in figure 16 the difference between both algorithms time performance is less than a factor of 3. Linear autosearch time function is beginning look better than linear the higher ntimes gets. Furthermore, for higher ntimes values, the functions become less irregular, but that is to be expected, as when more searches are measured, the approximation is more reliable.

5. Response to the theoretical questions.

Here, you answer the theoretical questions.

5.1 Question 1

In all the three functions the basic operation is the key comparison. In the linear searches it is an equality (table[i]==k) whereas in binary search, an inequality (greater or less).

5.2 Question 2

For Binary Search:

$W_{BS}(N)= \Theta(\log N)$, and $B_{BS}(N) = 2 = O(1)$

Best case of binary search is 2 key comparisons instead of 1 as seen in theory because we count the '>' and '<' comparisons as two.

For Linear Search:

$W_{BS}(N) = N = \Theta(N)$, and $B_{LS}(N) = 1 = O(1)$

5.3 Question 3

When the distribution of the keys is not uniform, linear auto search, given enough iterations, will sort the table in order of how frequent the keys are. Therefore, it will have on average less cost than Linear Search, because the most frequent keys that are searched are placed at the beginning of the array.

5.4 Question 4

We will assume that the algorithm has sorted the table by frequency, so now it is sorted (because the most frequent key is 1, then 2...).

$$K_{eg} = \left\lfloor \frac{1}{2} + \frac{\max}{1 + t \cdot \max} \right\rfloor, \quad N = \max, \; t \in [0,1]$$

↑ Uniformly distributed.

Let $K \in \{1, \dots, N\}$

$$K = \left\lfloor \frac{1}{2} + \frac{N}{1+tN} \right\rfloor \iff K \le \frac{1}{2} + \frac{N}{1+tN} < K+1.$$

$$\iff K - \frac{1}{2} \le \frac{N}{1+tN} < K + \frac{1}{2}$$

$$\iff \left(K - \tfrac{1}{2}\right)(1+tN) \le N \quad \wedge \quad \left(K + \tfrac{1}{2}\right)(1+tN) > N$$

$$\iff \left(K - \tfrac{1}{2}\right) tN \le N - K + \tfrac{1}{2} \quad \wedge \quad \left(K + \tfrac{1}{2}\right) tN > N - K - \tfrac{1}{2}$$

$$\iff t \le \frac{N - K + \frac{1}{2}}{\left(K - \frac{1}{2}\right) N} \quad \wedge \quad t > \frac{N - K - \frac{1}{2}}{\left(K + \frac{1}{2}\right) N}$$

$$\iff \frac{N - K - \frac{1}{2}}{\left(K + \frac{1}{2}\right) N} < t \le \frac{N - K + \frac{1}{2}}{\left(K - \frac{1}{2}\right) N}$$

But also, $t \le 1$. and

$\frac{N - K + \frac{1}{2}}{(K - \frac{1}{2}) \cdot N} < 1 \quad \forall N \in \mathbb{N} \Leftrightarrow K > 1$, so for $K = 1$,

$$p(1) = 1 - \frac{N - \frac{3}{2}}{N \cdot \frac{3}{2}} \xrightarrow{N \to \infty} 1/3$$

The probability of this happening is:

$$p(K) = \frac{N-k+1/2}{(k-\frac{1}{2})N} - \frac{N-k-\frac{1}{2}}{(k+\frac{1}{2})N} =$$

$$= \frac{(N-k+1/2)(k+\frac{1}{2}) - (N-k-\frac{1}{2})(k-\frac{1}{2})}{(k+\frac{1}{2})(k-\frac{1}{2})N}$$

$$= \frac{Nk + \frac{N}{2} - k^2 - \frac{k}{2} + \frac{k}{2} + \frac{1}{4} - Nk + k^2 - \frac{k}{2} + \frac{N}{2} + \frac{k}{2} - \frac{1}{4}}{(k+\frac{1}{2})(k-\frac{1}{2})N}$$

$$= \frac{1}{(k+\frac{1}{2})(k-\frac{1}{2})} = \frac{1}{k^2 - 1/4} = p(k), \; k \neq 1$$

Also, for a key K, with linear auto search, $n^S_{LAS}(K) = K$, because we are using an ordered array.

$$A^S_{LAS}(N) = \sum_{k=1}^{N} p(k) \cdot k = \frac{1}{3} \cdot 1 + \sum_{k=2}^{N} \frac{k}{k^2 - 1/4}$$

approx. by integrals

$$\simeq \frac{1}{3} + \int_1^N \frac{x}{x^2 - \frac{1}{4}} dx = \frac{\log\left(N^2 - \frac{1}{4}\right)}{2} + O(1)$$

$$= \log(N) + O(1).$$

So the average cost for linear auto search is ln(N)+O(1), which is of the order of the average of the binary search ($A_{BinarySearch}(N)=\Theta(\log(N))=\Theta(\ln(N))$, so the difference is only a multiplying constant that comes from the change o base of the logarithm):

$$A_{LinearAutoSearch}(N) = \Theta(A_{BinarySearch}(N)).$$

5.5 Question 5

Binary search is a recursive algorithm and therefore, we firstly have to assure that the recursive step simplifies the problem (reduces the size of the array where to seek the key) and that there is a base case where to stop.

For each case it compares the key to the element in the middle of the sorted table. If the key is equal, then the algorithm stops. If the key is bigger and is on the table, then it must be in the right half of the table, so it repeats the process in the right side. The same for the left side if the key is lower. However, if the key is not in the table, there will be a point where it must be higher than a value, and lower than the next, in which case it stops. Therefore, this algorithm works.

6. Conclusions.

After having measured the performance of the different algorithms, there are several aspects worth mentioning. To begin with, once more we have seen how inefficient is the linear search (cost O(N)) and its variation linear autosearch when the key distribution is proportional. On the other hand, binary search has a significant superior performance but a main drawback: it requires the table to be ordered. Moreover, if binary search is coded as a recursive function there would be an inherent risk of stack overflow.

However, when the distribution of keys is not proportional, linear autosearch becomes an alternative to take into account. Its efficiency compared with binary search will depend on the number of times each key is searched: with the potential distribution used in this practice the difference (in execution time) between linear auto search and binary search was only of a factor of 3 and what is more important, linear autosearch does not need an ordered table.