Analysis of Algorithms 2019/2020

Practice 1

Álvaro Castillo & Pablo Ernesto Soëtard, Group 1291

Code	Plots	Memory	Total

1. Introduction.

This first practice will serve as a settlement for future practices. It will be generated several .C files permutations.c, sorting.c, times.c that will be used as a library for the codes of the different C functions to be implemented; also it will be included in the .H files permutations.h,sorting.h, times.h the prototypes of the functions implemented in the .C files. In this practice we are also going to check the different run times of two algorithms comparing them to the theoretic results.

2. Objectives

The first 3 sections are related to the generation of disordered tables. Then number 4 and 6 request to think and create an algorithm to order them, while number 5 will measure the time that each algorithm takes to execute as well as other attributes.

2.1 Section 1

> We are going to create a function *int random num (int inf, int sup)* that that generates equiprobable random numbers between the integers inf, sup, both inclusive. We will have to think solutions for this problem as the % operator and the module are forbidden.

2.2 Section 2

> We have to implement a routine *int * generate perm (int N)*, that generates a permutation of a number (in an array) and then randomly order the array elements using the function created in Section 1 . The pseudocode is given so we will have to translate it to C language.

2.3 Section 3

> We will insert the routine *int ** generate permutations (int n perms, int N)* into the file which generates n perms equiprobable permutations of N elements using the function created in Section 2.

2.4 Section 4

> The implementation of the function *int InsertSort (int * table, int ip, int iu)* is required. We will have to understand how the algorithm InsertSort works in order to implement it.

2.5 Section 5

> We will have to implement the functions:
>
> s*hort average sorting time (pfunc order method, int n perms, int N, PTIME AA time);*

s*hort generate sorting times (pfunc order method, char * file,int num min, int num max, int incr, int n perms);*

*short save time table (char * file, TIME AA time aa, int n time)*

This functions will provide us the average clock time and the average, minimum and maximum number of times that the OB is executed in the execution of the sorting algorithm methods. As we execute the algorithm method several times we will save that features in a file. With this exercise we are able to measure the efficiency of each algorithm.

2.6 Section 6

We are required to implement a routine int *InsertSortInv (int * table, int ip, int iu).* The objective is to have understood InsertSort and be able to change the function, so it order the elements in reverse order. We also have to obtain the execution times and compare the results obtained with those of the InsertSort routine

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 Excel to plot the functions and gcc to compile all the files.

3.1 Section 1

The methodology of the random number function is described in the exercise 1. To check that it worked fine, we had executed that function a bunch of times, looking if all the numbers that we wanted, appeared and we have plotted a number histogram.

3.2 Section 2

Implementing the generate_perm(int N) function was an easy task, because we were given the pseudocode of the function. After testing it a few times, we noticed that everything worked as expected.

3.3 Section 3

As with section 2, implementing generate_permutations(int n_perms, int N) was an easy job. We only had to deal with some memory leaks, but after some code corrections, valgrind returned no errors.

3.4 Section 4

As in the theory lessons we have seen how the Insert Sort algorithm works, we could program it easily. Furthermore on the slides we have the pseudocode and the actual code of the algorithm.

3.5 Section 5

In this exercise we had to create 3 functions.

The first one, s*hort average sorting time,* saves in a structure data about an algorithm. Getting the number of basic operations was easy, but we had a problem calculating the average number of B.O.

To do this, first we called the method n times and then we added the number of B.O and divided it by n. But when we had functions with large permutations that had a big number of Key comparisons our program overflowed and gave us negative numbers. So to fix this, we had calculated the solutions partially and added them.

Other problem that we had was related with measuring the time. We have used the clock_gettime() function, so we cannot compile the exercise with -ansi as it is not included in this library. As none had explained to us how that function worked, we had copied it from the internet:

https://users.pja.edu.pl/~jms/qnx/help/watcom/clibref/qnx/clock_gettime.html

When we compiled it and executed it, we noticed that it gave us some negatives values and obviously negative seconds didn't make any sense. Therefore, we looked for a solution on the Internet and we found this:

```
if ((stop.tv_nsec-start.tv_nsec)<0){

        ptime->time = (double)((double)((stop.tv_sec - start.tv_sec-1)+(1000000000 +
stop.tv_nsec - start.tv_nsec)) / BILLION ) /n_perms;

    }else{

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

    }
```

We have then executed the function another time, and we had no more negative values, but it gave us random times, so we thought that, that solution wasn't a great one. Then we asked some colleagues and they said to us that we should cast each of the variables and operations that we do to calculate the time. Having done such thing and having executed again the function it finally gave us correct results.

The second function was s*hort generate sorting times. W*e allocate memory for all the data structures of PTIME. That means allocate (num_max-num_min)/incr +1 structures to save all data.

The third function *short save time table*, was easy to program as it prints in a file the information of the structure.

3.6 Section 6

As we knew how the InsertSort worked, then we just had to copy the function and change one comparison, to make InsertSortInv work. Instead of looking if the element T[j] is greater than T[i], we look if it is less than T[i].

4. Source code

4.1 Section 1

```
int random_num(int inf, int sup){
  if(inf>sup){
    return -1;
  }
  return (int)((sup - inf + 1.0)*rand()/(RAND_MAX + 1.0)+ inf);
}
```

4.2 Section 2

```
int* generate_perm(int N){
    int i=1, a=0, random=0;
    int *perm;
    if(N<1){
        return NULL;
    }
    perm = (int*)malloc(N*sizeof(int));
    if(!perm){
        return NULL;
    }
    for(;i<=N;i++){
        perm[i-1] = i;
    }
    for(i=0;i<N;i++){
        a = perm[i];
        random = random_num(i, N-1);
        if(random==-1){
            free(perm);
            return NULL;
        }
        perm[i] = perm[random];
        perm[random] = a;
    }
    return perm;
}
```

4.3 Section 3

```
int** generate_permutations(int n_perms, int N){
    int i,j;
    int **perms;
    if(n_perms<1 || N<1){
        return NULL;
    }
    perms = (int**)malloc(n_perms*sizeof(int*));
    if(!perms){
        return NULL;
    }
    for(i=0; i<n_perms; i++){
        perms[i] = generate_perm(N);
        if(!perms[i]){
            for(j=0; j<i; j++){
                free(perms[j]);
            }
            free(perms);
            return NULL;
        }
    }
    return perms;
}
```

4.4 Section 4

```
int InsertSort(int* list, int ip, int iu){
    int i, j, a,counter=0;

    if(!list || ip<0 || iu<ip){
        return ERR;
    }

    for(i=ip+1; i<=iu; i++){
        a = list[i];
        j = i-1;
        while(j>=ip && list[j]>a){
            counter++;
            list[j+1]=list[j];
            j--;
        }
        if(j>=ip){
            counter++;
        }
        list[j+1] = a;

    }

    return counter;
}
```

4.5 Section 5

```
short average_sorting_time(pfunc_sort method, int n_perms, int N, PTIME_AA ptime){

    int i, **perms, res;
    struct timespec start, stop;

    if(!method || n_perms<0 || N<0 || !ptime){
        return ERR;
    }
    ptime->average_ob=0;
    ptime->N=N;
    ptime->n_elems=n_perms;

    perms = generate_permutations(n_perms, N);
    if(perms==NULL){
        return ERR;
    }

    clock_gettime(CLOCK_REALTIME, &start);


    for(i=0; i<n_perms; i++){
        res = method(perms[i], 0, N-1);
        if(res==ERR){
            clock_gettime(CLOCK_REALTIME, &stop);
            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_perms;

    }
    clock_gettime(CLOCK_REALTIME, &stop);


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

    for(i=0; i<n_perms; i++){
        free(perms[i]);
    }
    free(perms);
    return OK;
}
```

```
short generate_sorting_times(pfunc_sort method, char* file,  int num_min, int num_max,  int incr, int n_perms){
    PTIME_AA ptime=NULL;
    int i,j;
    int n_times;

    if(!method || !file || num_min<0 || num_max<num_min || incr<1 || n_perms<0){
        return ERR;
    }

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

    ptime = (PTIME_AA)malloc(n_times*sizeof(TIME_AA));

    if(!ptime){
        return ERR;
    }

    for(i=num_min, j=0;i<=num_max;i+=incr, j++){
        if(average_sorting_time(method, n_perms, i, &ptime[j])==ERR){
            free(ptime);
            return ERR;
        }
    }

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

    return OK;
}
```

```c
short save_time_table(char* file, PTIME_AA ptime, int n_times){

    FILE *f;
    int i;

    if(!file || !ptime || n_times<0){
        return ERR;
    }

    f=fopen(file, "w");
    if(!f){
        return ERR;
    }

    for(i=0; i<n_times; i++){
        fprintf(f,"%d %lf %lf %d %d\n", ptime[i].N, ptime[i].time, ptime[i].average_ob, ptime[i].max_ob,
        ptime[i].min_ob);
    }

    fclose(f);

    return OK;
}
```

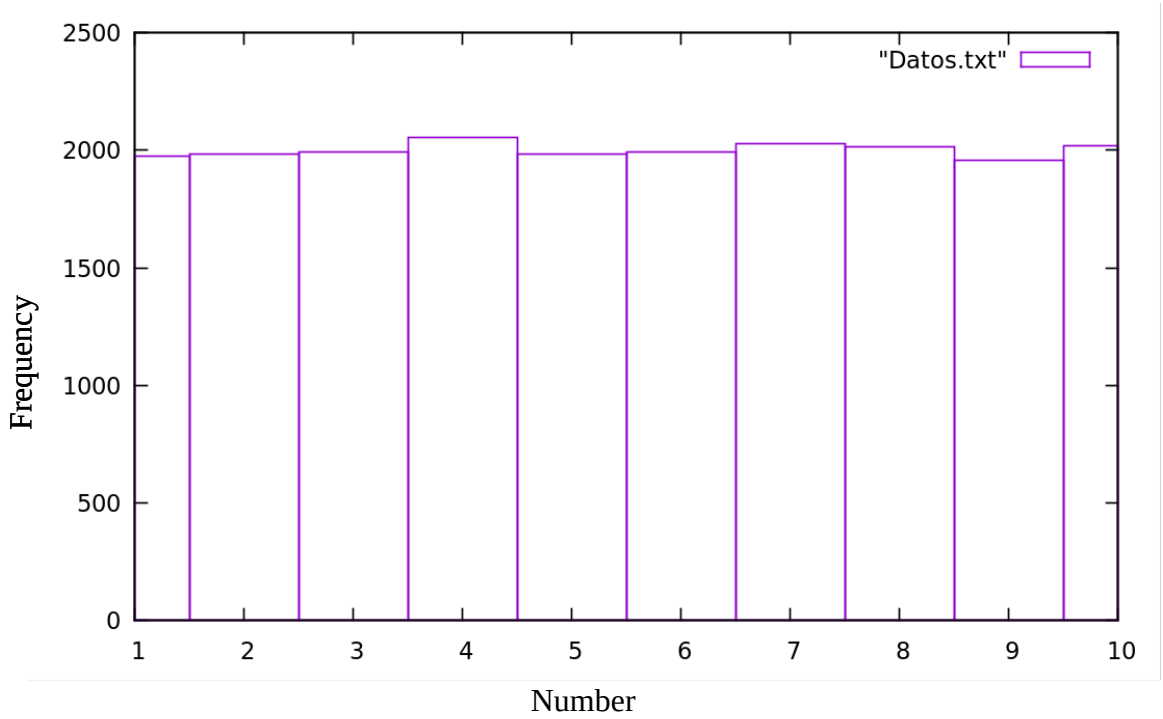
4.6 Section 6

```c
int InsertSortInv(int* list, int ip, int iu){
    int i, j, a,counter=0;

    if(!list || ip<0 || iu<ip){
        return ERR;
    }

    for(i=ip+1; i<=iu; i++){
        a = list[i];
        j = i-1;
        while(j>=ip && list[j]<a){
            counter++;
            list[j+1]=list[j];
            j--;
        }
        if(j>=ip){
            counter++;
        }
        list[j+1] = a;
    }

    return counter;
}
```

5. Results, Plots

5.1 Section 1

We have checked that our random number generation function works properly



Here we have the plot of the random number histogram.

We have introduced into the `random_num` function a minimum value of 1 and a maximum value of 10 and we have called this function 20.000 times to check if it's behavior is correct.

As the boxes histogram shows, the times that a random number is generated is almost the same for each number. There is a bit difference between the results in the number of times they appear, but if we increase the number of calls to the function, this difference will be less and there will be a moment where all results will be the practically the same.

This means that the algorithm we have proposed to solve the random number problem works correctly and it has a great level of randomness.

5.2 Section 2

We have used the file exercise2.c to check the correct functioning of the routine. As we can see when we execute that file, the function that we have created correctly generates random permutations.

5.3 Section 3

We have used the file exercise3.c to check the correct functioning of the routine. If we execute it, it gives us n-perms of equiprobable permutations of N elements

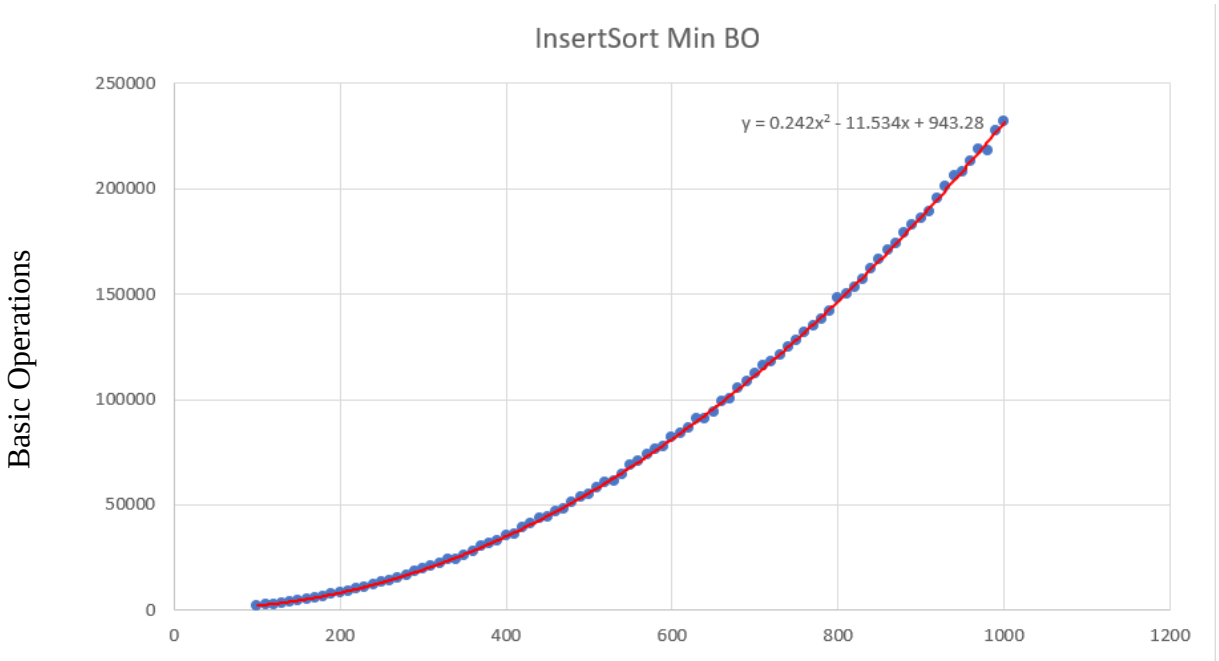
5.4 Section 4

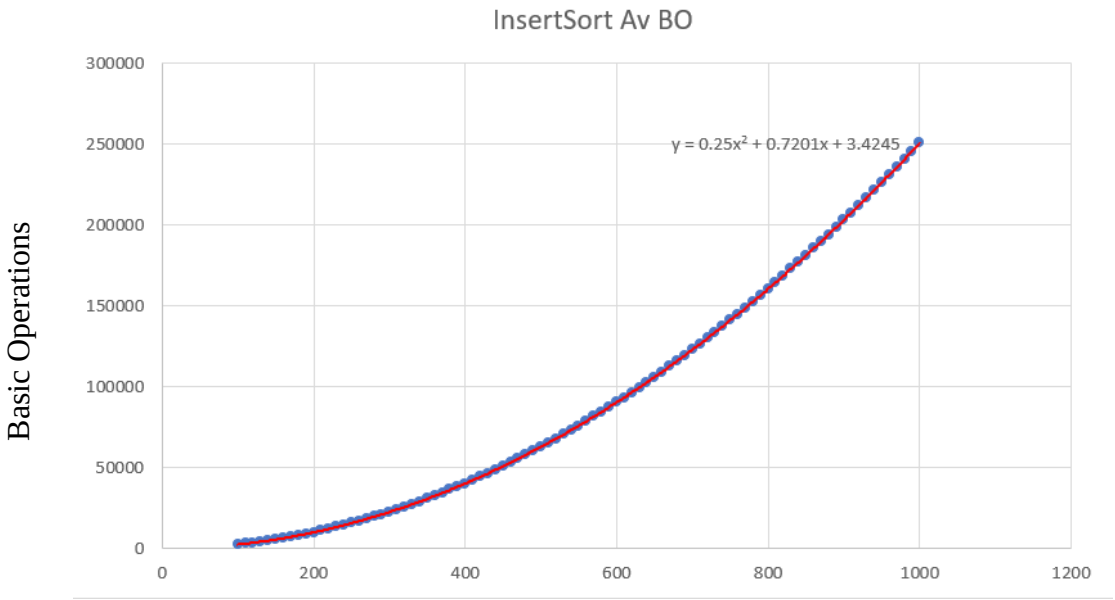
We have used the file exercise4.c to check the correct functioning of the InsertSort algorithm. If we execute it several times we can check that the algorithm correctly sorts all the combinations of different permutations.

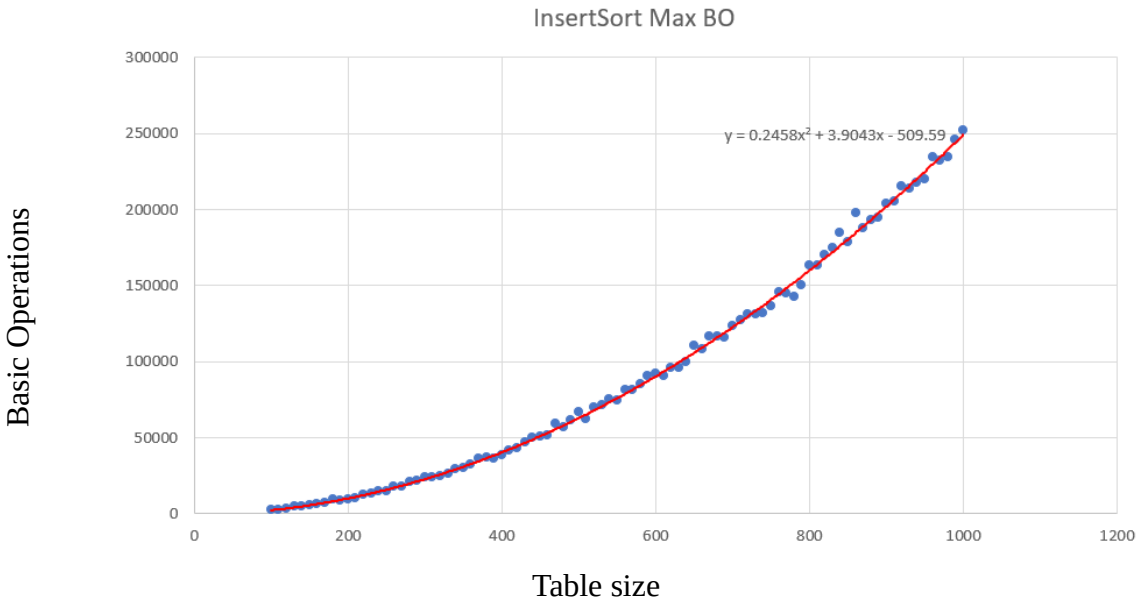
5.5 Section 5

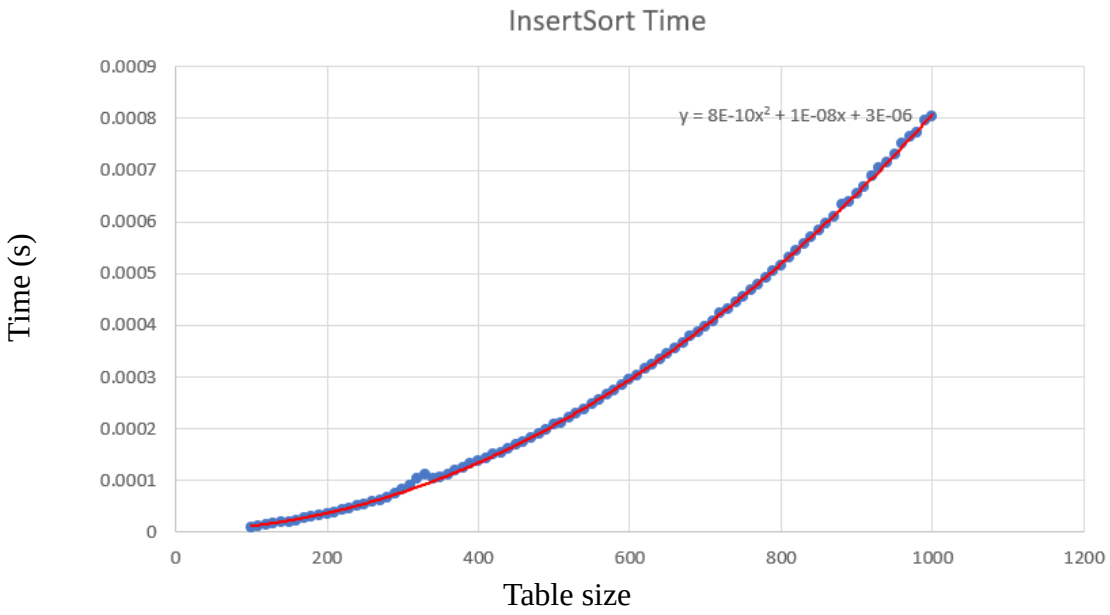
To check the correct functioning of the functions implemented until now, we have executed the file exercise5.c giving us positive results.

We have generated some plots using 10000 permutations for each size, taking sizes from 100 to 1000 in increments of 10.




InsertSort Av BO
Basic Operations
y = 0.25x² + 0.7201x + 3.4245
300000
250000
200000
150000
100000
50000
0
0
200
400
600
800
1000
1200
Table size


InsertSort Max BO
Basic Operations
y = 0.2458x² + 3.9043x - 509.59
300000
250000
200000
150000
100000
50000
0
0
200
400
600
800
1000
1200
Table size



In those images we can see the plots of the Min BO, Average BO, Max BO and Average Time of the Insert Sort algorithm. The X axis represents the size of the array passed to the Insert Sort algorithm, and the Y axis is the number of basic operations in the case of the Min, Average and Max plot, and seconds for the Time plot.

With a red line, a quadratic polynomial that fits the values is represented, and with blue dots the data retrieved data from our program is showed.

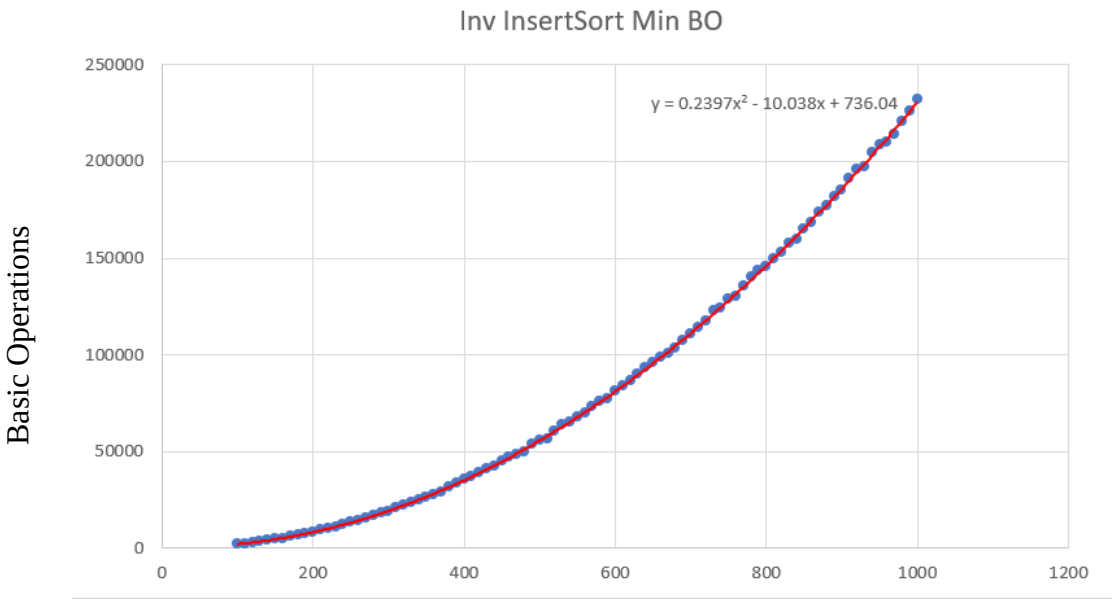
We have noticed that the values that we have plotted fitted perfectly with a N^2 function in all plots. That is what was expected for the Average OB($N^2/4$), Max OB ($N^2/4$) and Time ($a*N^2$), but not for the Min OB, which theoretical value is N-1. After trial and error we came to the conclusion that the results that we got from our program were caused due to the low probability of getting a sorted array from the generate_permutations() function, since the probability to get a sorted array from it is (1/n!) which get lower and lower while we increase the size of the array.

Therefore, having that implementation of the generate_permutations() function, it is normal to get a quadratic polynomial as the Min BO.

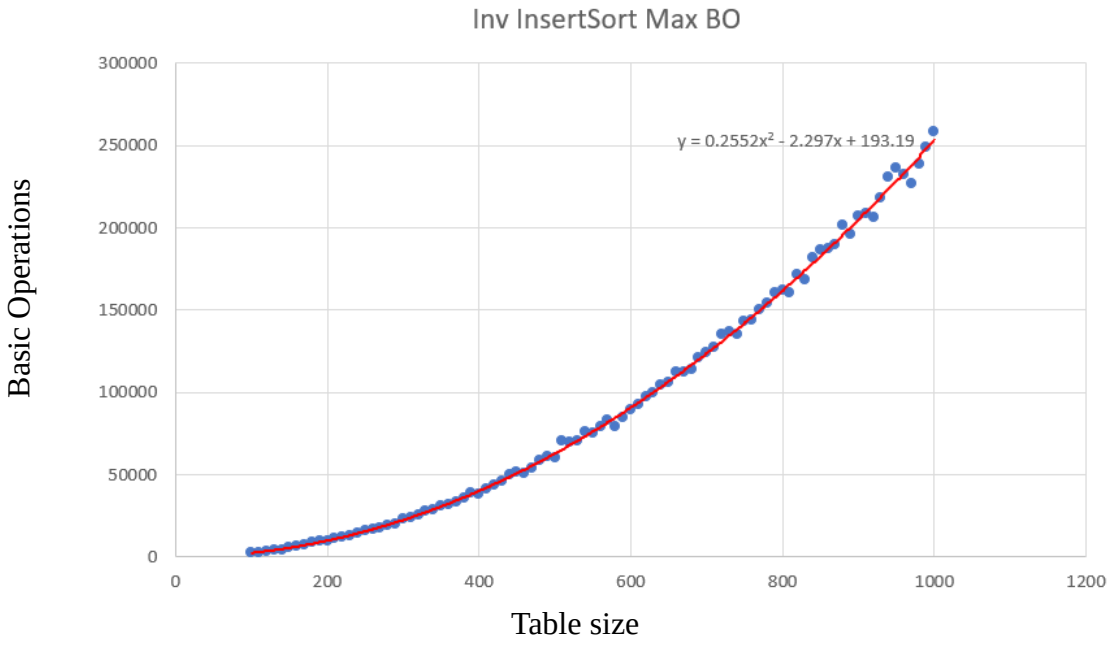
5.6 Section 6

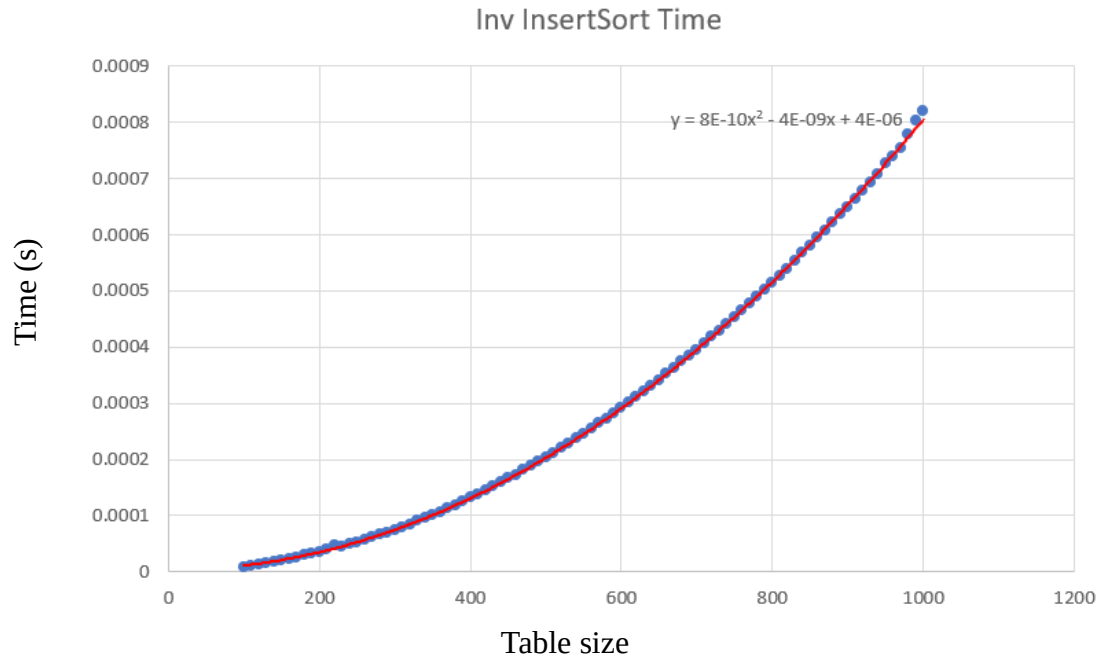
To check the correct functioning of the function InsertSortInv, we have executed the file exercise5.c giving us positive results.

We have generated some plots using 10000 permutations for each size, taking sizes from 100 to 1000 in increments of 10..


Inv InsertSort Min BO
y = 0.2397x² - 10.038x + 736.04
250000
200000
150000
100000
50000
0
0
200
400
600
800
1000
1200
Basic Operations
Table size


Inv InsertSort Av BO
y = 0.25x² + 0.7425x - 0.0519
300000
250000
200000
150000
100000
50000
0
0
200
400
600
800
1000
1200
Basic Operations
Table size





In those images we can see the plots of the Min BO, Average BO, Max BO and Average Time of the Insert Sort Inverse algorithm. The X axis represents the size of the array passed to the Insert Sort Inverse algorithm, and the Y axis is the number of basic operations in the case of the Min, Average and Max plot, and seconds for the Time plot.

As we can see from the above plots, and as it was expected, the values retrieved from the InsertSortInv algorithm are very close to the ones from the Insert Sort algorithm discussed earlier.

6. Answers to theoretical Questions.

6.1 To implement the random number function we have thought at first using the module of the range that the function receives and the srand functionality. Per example:

```
srand(time(NULL));

number= rand () % (sup - inf +1) + inf;
```

But our teacher said that, that way of generating a random number is not equiprobable and that neither the modulo or % operator cannot be used.

Then he explained the way we might do it. We have to adapt our range to another one that goes from 0 to 1 dividing by RAND_MAX+1 and then we scale it by a factor of sup-inf+1 and add our minimum number. The function should be this one:

```
((sup - inf + 1.0)*rand()/(RAND_MAX + 1.0)+ inf)
```

Another method for a random generation could be:

```
 number= inf+ (rand() % sup - inf);
```

This method is extracted from the book Numerical Recipes and it is not as equiprobable as the one we have used.

6.2 The algorithm InsertSort just compares a number in a specific position in the array with its previous ones. If it is less than it's previous, then the algorithm changes it and keep comparing it until the previous number is greater than the one we want to order. It works properly because it starts from the beginning (2º number in the array) sorting the numbers one by one until it gets the end of the array.

6.3 The outer loop of InsertSort do not act on the first element of the table because this algorithm assumes that the first element is already sorted. Therefore it does not need to check if it's previous element is less than him, and obviously this operation does not makes sense, as the first element has not a previous one.

6.4 The basic operation should be in the innermost loop and it must be representative of the algorithm. In the InsertSort algorithm the B.O is the comparison : `list[j+1]<list[j]` as it fulfills both requests.

6.5 As expected, the plots of InsertSort and InsertSortInv are almost identical. This is because both algorithms perform the same task (sort an array) and have almost the same code, just differing in a "<" instead of an ">".

7. Final Conclusions.

Working on this practice, we have understand more deeply how local sorting algorithms work.

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 memory leaks and coding errors or bugs, 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 learn how to use new tools such as Excel, as well as overcoming some problems that appeared during the process.