Analysis of Algorithms 2020/2021

Practice 1

Pablo Cuesta Sierra, Álvaro Zamanillo Sáez. Group 1251.

Code	Plots	Memory	Total

1. Introduction.

In this practice we will compare the number of basic operations carried out by two algorithms that sort a given table of integers: insert sort and insert sort inverse.

In order to do so, we have created several auxiliar functions to generate the 'random' permutations to sort. The code of these functions, algorithms and the results are shown below.

2. Objectives

2.1 Section 1

We try and look at the effectiveness of the function 'random_num' that we have designed. We also discuss whether it is necessary to find an alternative to 'rand()%size'.

2.2 Section 2

We create a routine that allocates a random permutation of N (number given).

2.3 Section 3

We use the previous routine to make a function that creates a given number of permutations of a given size.

2.4 Section 4

This consists of the implementation of the algorithm InsertSort, which returns the number of times that the basic operation of the algorithm (key comparison) has been performed for the given input.

2.5 Section 5

We implement the tools that we later use to analyze the efficiency of the algorithms.

2.6 Section 6

In this section we compare the results obtained from executing the two functions: InsertSort and InsertSortInv. We also compare these results with the expected theoretical values.

3. Tools and Methodology

Here you describe the environment (Windows, Linux, MacOS) the tools you have used (Netbeans, Visual Studio, Eclipse, gcc, Valgrind, Gnuplot, Sort, uniq, etc) and the development methodologies and solutions to the problems that you have used in each section, so as the tests you have done to the developed programs.

We have developed the whole practice in Linux (Ubuntu, with the GCC version 9.3.0) using Visual Studio Code (with the extension Live share so we could work at the same time). For plotting the results, we have chosen Google Spreadsheets and Gnuplot.

3.1 Section 1

In a first approach we just used the random function with module operator. It was not as random as requested but it allowed us to continue with the rest of the exercises while we read about the topic to find new ways of generating 'random' numbers.

In a second approach we discarded the module operator, but we still did use of rand() to create a parameter 't'. (More detail is given in Q1, section 6.1).

3.2 Section 2

Here we had to be careful when allocating memory, to free it whenever there were any errors, this function was very simple.

3.3 Section 3

The function for this exercise made use of the previous function in order to make multiple permutations of a determined size.

3.4 Section 4

Here we implemented the algorithm InsertSort and the main concern was to be able to correctly count the number of times that the key comparison was performed each time that the algorithm was called. In the end we made sure that all the cases were covered by trying for multiple values and inputs. The results of this are shown in the plots for the minimum and maximum number of BO that are in section 5.5.1 of this report, where for small values (the ones that can be fully covered in a reasonable amount of time), the values for best and worst case are the same as the ones calculated in the theory classes.

3.5 Section 5

In this part we encountered three main problems:

The first one was that we could not measure the average time of the algorithms due to precision problems with the library "time.h". That was finally solved when we started using the function 'gettimeofday', from the C library 'sys/time.h'.

Secondly, when executing the exercise to test and running valgrind we noticed that we wrote in not reserved memory. After reviewing the code several times, we realized

we were not allocating enough memory for the PTIME_AA array, but one element less, easily fixed by increasing the size of the array to be reserved.

Finally, when collecting the results for big numbers, we saw that the size of the type of variable used in the structure PTIME for storing the minimum and maximum values was not big enough, specifically for permutations of size 95000 and 100000. We could not solve this inconvenience, as we would have had to change the structure that was given. We, however, changed the type of the variable in which we stored the total amount of BO (used later to calculate the average)-to 'unsigned long', -because there was an overflow in this variable for big sizes of permutations, as it stored bigger numbers than other variables as min or max.

3.6 Section 6

After having coded InsertSort, we had not much of a problem with InsertSortInv, as the idea was basically the same. The only thing that changes in the code is the key comparison which changes from: num<table[j]; to: num>table[j].

4. Source code

Here you write the source code **only the routines you have developed** in each section.

4.1 Section 1

```
/*************************************************/
/* Function: random_num Date:   2020/10/16       */
/* Authors: Álvaro Zamanillo and Pablo Cuesta     */
/*                                               */
/* Rutine that generates a random number         */
/* between two given numbers                     */
/*                                               */
/* Input:                                        */
/* int inf: lower limit                          */
/* int sup: upper limit                          */
/* Output:                                       */
/* int: random number                            */
/*************************************************/
int random_num(int inf, int sup){
 double t;

 if(inf>sup){
  return ERR;
 }

 t=(double)rand() / ((double)RAND_MAX+1.);

 return inf+((int)(t*(sup-inf+1)));
}
```

4.2 Section 2

```c
/*****************************************************/
/* Function: generate_perm Date:  2020/09/23         */
/* Authors: Álvaro Zamanillo and Pablo Cuesta        */
/*                                                   */
/* Rutine that generates a random permutation        */
/*                                                   */
/* Input: Pablo Cuesta, Álvaro Zamanillo             */
/* int n: number of elements in the permutation      */
/* Output:                                           */
/* int *: pointer to integer array                   */
/* that contains the permitation                     */
/* or NULL in case of error                          */
/*****************************************************/
int* generate_perm(int N){

  int *perm=NULL;
  int i,temp, num;

  if(N<=0)
    return NULL;

  if(!(perm=(int*)calloc(N,sizeof(int))))
    return NULL;

  for(i=0;i<N;i++){
    perm[i]=i+1;
  }

  for(i=0;i<N;i++){
    num = random_num(i,N-1);
    temp=perm[i];
    perm[i]=perm[num];
    perm[num]=temp;
  }

  return perm;
}
```

4.3 Section 3

```c
/*************************************************/
/* Function: generate_permutations Date: 2020/09/23*/
/* Authors: Pablo Cuesta, Álvaro Zamanillo         */
/*                                                 */
/* Function that generates n_perms random          */
/* permutations with N elements                    */
/*                                                 */
/* Input:                                          */
/* int n_perms: Number of permutations             */
/* int N: Number of elements in each permutation   */
/* Output:                                         */
/* int**: Array of pointers to integer that point  */
/* to each of the permutations                     */
/* NULL en case of error                           */
/*************************************************/
int** generate_permutations(int n_perms, int N)
{

  int **perm=NULL;
  int i,j;

  if(n_perms<=0||N<=0)
    return NULL;

  if(!(perm=(int**)calloc(n_perms,sizeof(int*))))
    return NULL;

  for(i=0;i<n_perms;i++){
    perm[i]=generate_perm(N);
    if(!perm[i]){
      for(j=0;j<i;j++)
        free(perm[j]);
      free(perm);
      return NULL;
    }
  }

  return perm;
}
```

4.4 Section 4

```
/************************************************/
/* Function: InsertSort    Date:   2020/10/14      */
/* Implementation of the algorithm InsertSort      */
/* *table: array to be sorted                      */
/* ip: first index of the array                    */
/* iu: last index of the array                    */
/************************************************/
int InsertSort(int* table, int ip, int iu)
{ int i = ip+1, j, num, count=0;

  if(!table||ip<0||iu<ip)
    return -1;

  while (i<=iu){
    num=table[i];

    for(j = i-1;j>=ip && num<table[j]; j--){
      count++;
      table[j+1] = table[j];
    }
    if(j>=ip){
      count++;
    }

    table[j+1]=num;
    i++;
  }

  return count;
}
```

4.5 Section 5

```
/**********************************************/
/* Function: average_sorting_time Date: 2020/09/30 */
/*
 * This function stores the data received from
 * sorting n_perms permutations of size N
 *
 * method: sorting method to be used
 * n_perms: represents the number of permutations to be generated
 *      and sorted by the used method (in this case insertion method),
 * N: is the size of each permutation
 * time: pointer to a type structure TIME AA that at the exit of the func
tion will contain:
 * the number of permutations averaged in the n elems field,
 * the size of the permutations in the N field,
 * the average execution time (in seconds) in the time field,
 * the average number of times the OB was executed in the average ob fiel
d,
 * the minimum number of times the OB was executed in the min ob field
 * the maximum number of times the OB was executed in the max ob field
 *
 **********************************************/
short average_sorting_time(pfunc_ordena metodo,int n_perms,int N, PTIME_A
A ptime)
{
    struct timeval start, end;
    long micro;
    double elapsed;

    int **perms=NULL, i=0, min, max;
    unsigned long c_total=0, c_aux=0;

    if(!metodo||n_perms<=0||N<=0||!ptime){
        return ERR;
    }
    if(!(perms=generate_permutations(n_perms, N))){
        return ERR;
    }


    gettimeofday(&start,NULL);
```

```
    for (i=0;i<n_perms && c_aux!=-1;i++){
        c_aux=(unsigned long)metodo(perms[i], 0, N-1);

        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;
    }

    gettimeofday(&end,NULL);
    elapsed=(end.tv_sec-start.tv_sec);
    micro=(end.tv_usec-start.tv_usec)*0.000001;
    elapsed+=micro;

    int_pp_free(perms, n_perms);

    if(c_aux==-1){
        return ERR;
    }

    ptime->time = (double)elapsed/(double)n_perms;
    ptime->average_ob = (double)c_total/(double)n_perms;
    ptime->max_ob = max;
    ptime->min_ob = min;
    ptime->N = N;
    ptime->n_elems = n_perms;

    return OK;
}
```

```c
/***************************************************/
/* Function: generate_sorting_times Date: 2020/09/30
 *
 * this function calls average_sorting_time to test
 * and then print the information that said function provides
 * for permutations of sizes from num_min to num_max (+incr)
 *
 * method: function used to sort
 * file: name of the file where to print the information
 * num_min: minimum size of the permutations
 * num_max: maximum size of the permutations
 * incr: increment of the size
 * n_perms: number of permutations of each size
 *
 * ***************************************************/
short generate_sorting_times(pfunc_ordena method, char* file, int num_min
, int num_max,
    int incr, int n_perms)
    {
    int N, i, n_iter;
    TIME_AA *ptime=NULL;

    if(!method||!file||num_min<=0||num_max<=num_min||incr<=0||n_perms<=0)
        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_sorting_time(method, n_perms, N, ptime+i)==ERR){
            free(ptime);
            return ERR;
        }
        printf("Num %d\n",N);
        i++;
    }

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

    return OK;

}
```

```
/*************************************************/
/* Function: save_time_table Date:   2020/09/30
 *
 * This is the function called by generate_sorting_times
 * to print the information stored in the array ptime
 * n_times is the size of the array and file, the name
 * of the file where the data is to be printed
 *
 *************************************************/
short save_time_table(char* file, PTIME_AA ptime, int n_times)
{
   FILE* pf;
   int i;

   if(!ptime||n_times<1||!file)
        return -1;

    if(!(pf=fopen(file,"a")))
        return -1;

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

    fclose(pf);

    return OK;

}
```

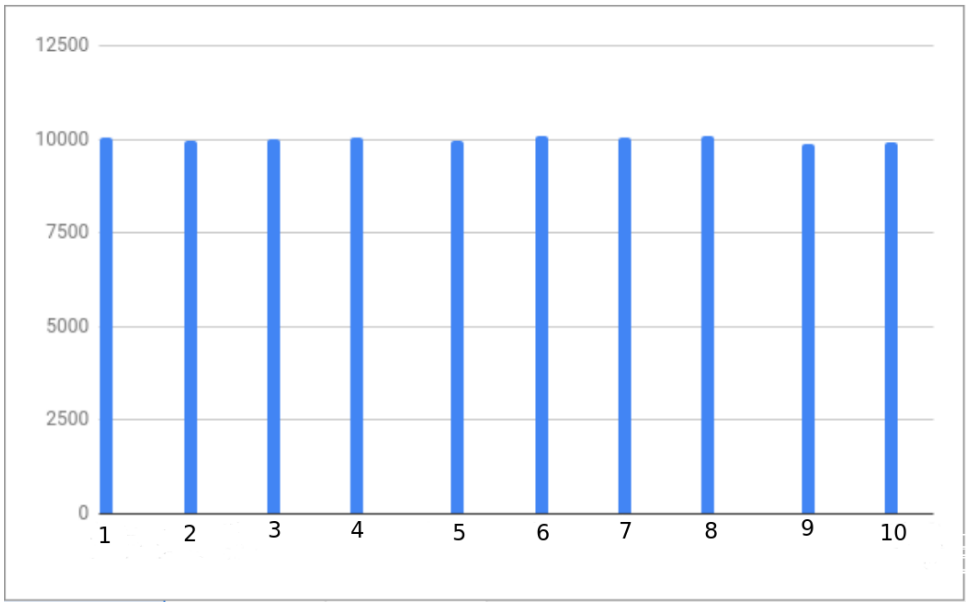
Section 6

```
/************************************************/
/* Function: InsertSortInv    Date:   2020/10/14   */
/* Implementation of the algorithm InsertSort to   */
/* sort an array in descending order               */
/* *table: array to be sorted                      */
/* ip: first index of the array                    */
/* iu: last index of the array                     */
/************************************************/
int InsertSortInv(int* table, int ip, int iu)
{ int i = ip+1, j, num, count=0;

  if(!table||ip<0||iu<ip)
    return -1;

  while (i<=iu){
    num=table[i];

    for(j = i-1;j>=ip && num>table[j]; j--){
      count++;
      table[j+1] = table[j];
    }
    if(j>=ip){
      count++;
    }

    table[j+1]=num;
    i++;
  }

  return count;
}
```

5. Results, Plots

5.1 Section 1

This is the result of executing exercise 1. With the random function that we have created. We generated for this plot 100,000 numbers from 1 to 10. The results are quite equally distributed, as intended. Although this generator is not completely random (it is pseudo-random), it is enough for the tasks we want it for. Comments on the function chosen for this purpose are in section 6.1.



5.2 Section 2

In this section we successfully get permutations of the natural number that is input in the function generate_perm.

5.3 Section 3

The function generate_permutations returns an array of arrays of the permutations requested without any errors or memory leaks.

5.4 Section 4

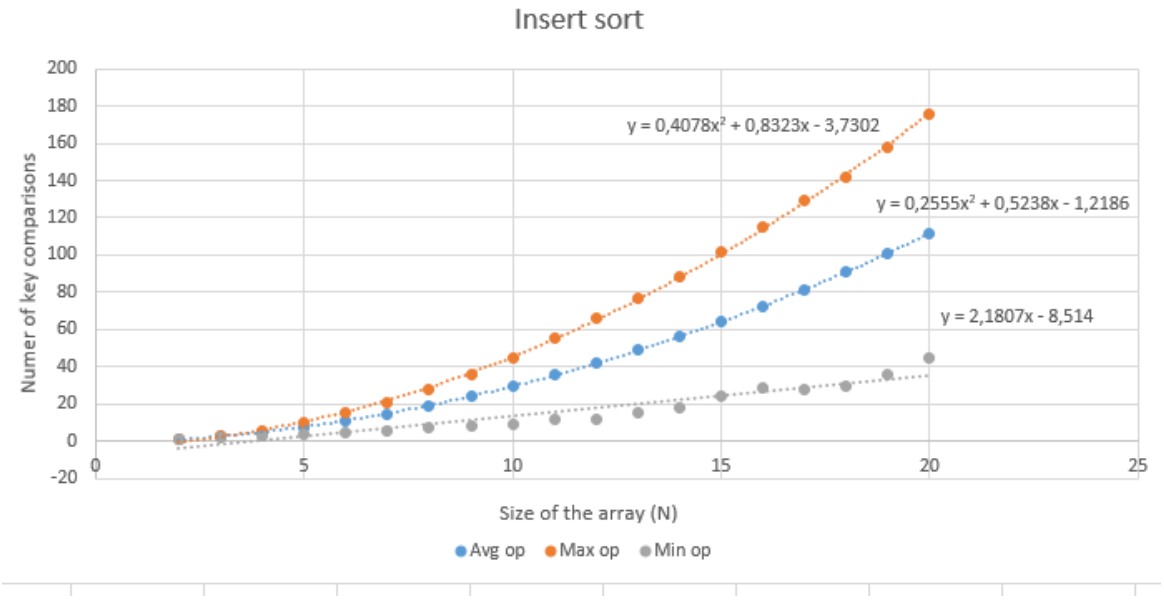
Here we make sure that the function correctly sorts permutations with no errors.

5.5 Section 5

In order to collect the data to compare the results with the theoretical values, we have separated our data collection in two parts. First, we have collected the data of sorting 10^6 permutations of sizes 1 to 20, in order to look at the worst and best cases. And then we have collected data of sorting 200 permutations of sizes 10^4 to 10^5 with an increment of 5,000, so that we could measure the average times, because for low values, time could not be measured by the program as it was very small.

5.5.1 Best and Worst cases.

Here is the plot of the average time with respect to N, the size of the permutations of the data we collected:

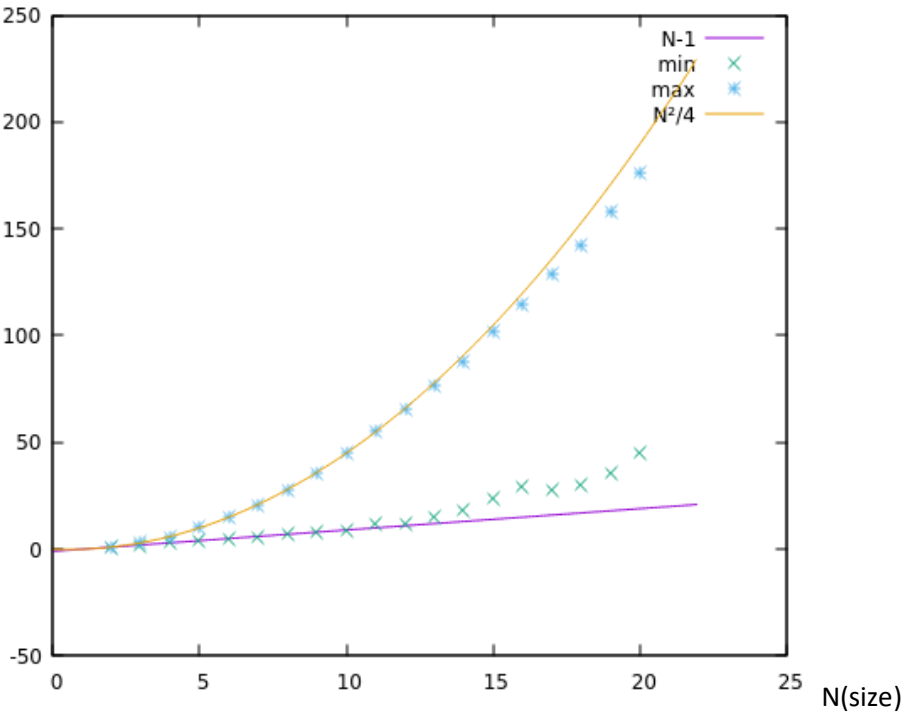


As the plot shows, the maximum number grows as a function that is $0.40*N^2+O(N)$, according to the approximation done by excel (we are not considering more than 2 decimals from the equation given by excel). This result almost matches with the expected theoretical result, $0.50*N^2+O(N)$.

When it comes to the average operation, the function is, in this case $0.25N^2+O(N)$. This time it does match with the theoretical result.

Finally, in the minimum number of operations we see that it is not ~N (asymptotically equivalent to N) as the theoretical result (which is N-1), but $\Theta(N)$. For low values of N, from 1 to 10, the minimum number of operations is in fact N-1, however for higher values it deviates. This is because we have not taken enough permutations: for N=10, we have 10! different permutations, that is 3.6 million permutations; we have only considered 1 million (and random). For N=20, we have 20! (in the order of 10e8) different permutations and so on. It is clear, that the chance to get the ordered permutation is very low for high values of N, thus, the difference between the theoretical result and the obtained.

Here are the results compared to the estimated functions calculated in theory (x axis has the size of the permutation, y axis represents the max/min number of BOs):

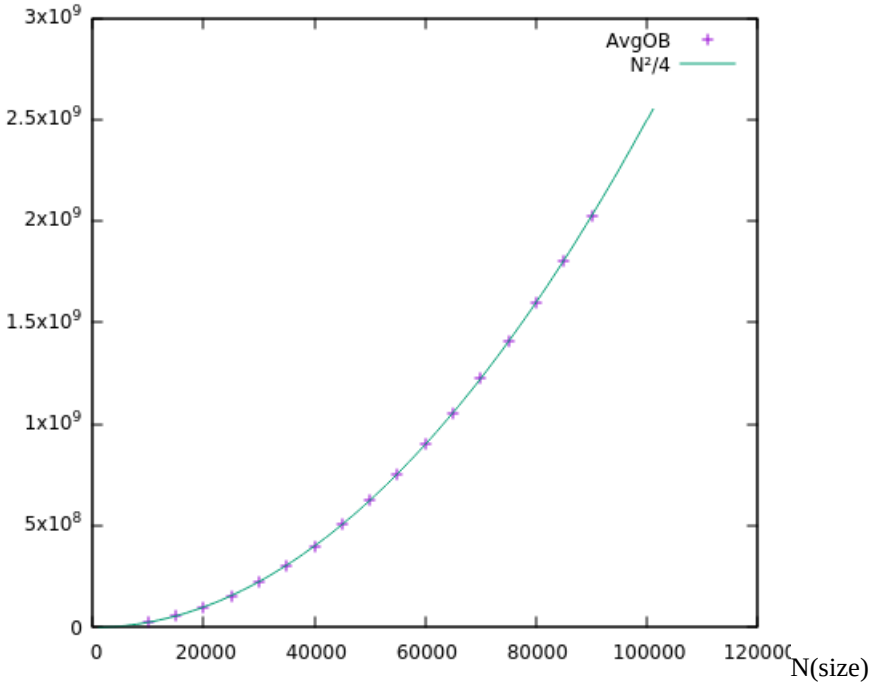


5.5.2 Average case

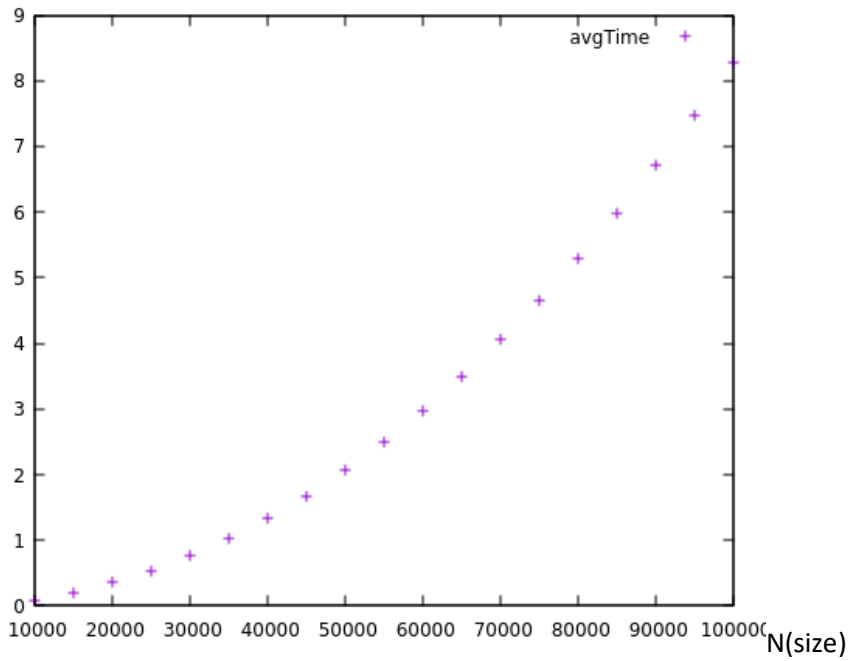
As we have seen in the theory class, the function of the average case for insert sort for the algorithm InsertSort with an input of size N is asymptotically equivalent to $N^2/4$. That is, $A_{InsertSort}(N)=N^2/4+O(N)$.

This is the data collected by sorting 200 permutations, compared with $N^2/4$ we see that it is almost the same:

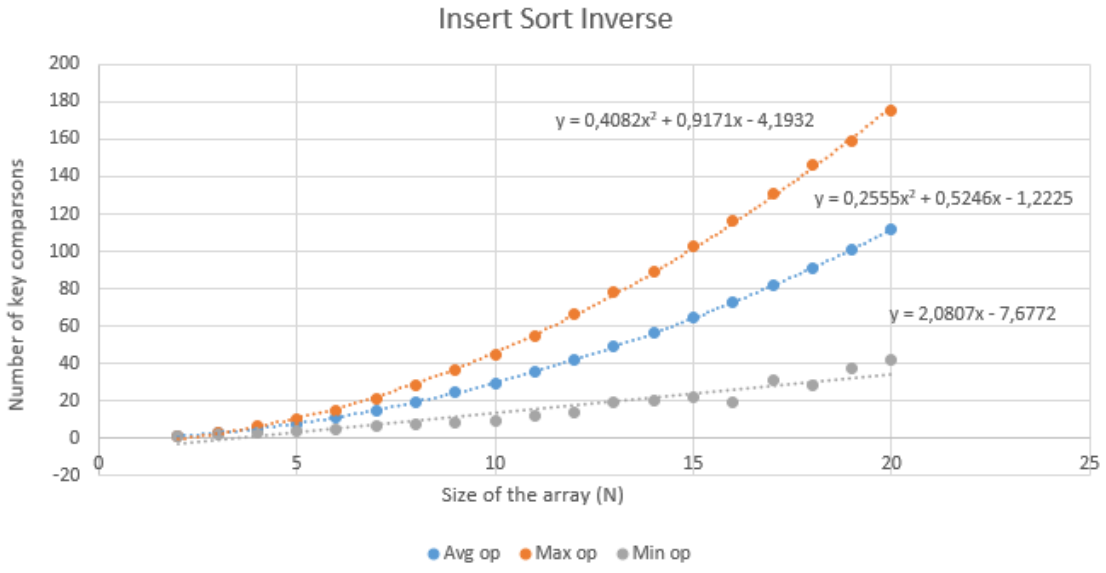
Average Basic Operation with InsertSort:



Average Time with InsertSort (we can see that it is a quadratic function-as expected), with time in the y axis, in seconds, and N in the x axis (number of permutations):



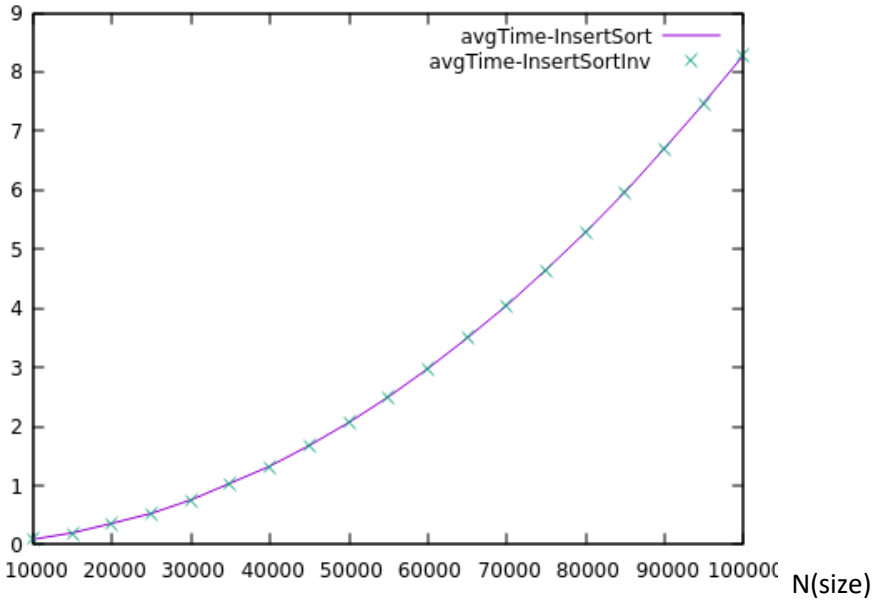
5.6 Section 6



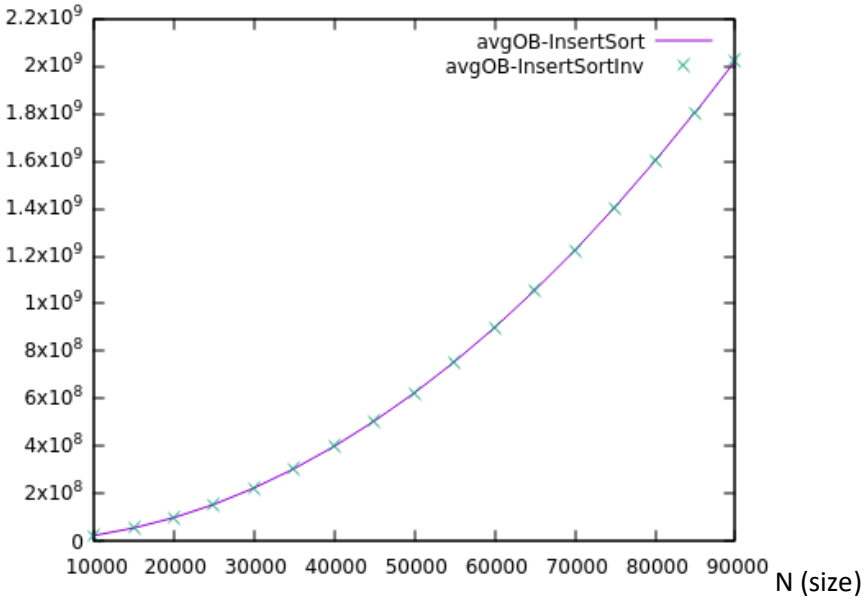
The results obtained with this algorithm are exactly the same as the ones collected with the other one. This is because the algorithms do the same, the only difference is that one checks if a value is below another in order to sort and the other one checks if the value is above to sort.

Once more the data obtained does not fit in the case of the minimum number of operations for the same reason as before.

Here we see the plot of the times collected with InsertSortInv alongside those of InsertSort for 200 permutations of size 10^4 to 10^5. Time is in seconds and one of them is represented without connecting lines as the two plots would be indistinguishable otherwise, because the results are (almost exactly) the same.



The same happens with the plot of the average of the basic operation:



6. Answers to theoretical Questions.

6.1 Question 1

Our first approach included the function modulus (%), however this is not strictly random because of the following:

Rand gives a number in the interval [0, RAND_MAX], and, although it is not entirely random, it is pseudo-random (which would be a problem if this is used for any security-related programs), every number in this interval has the same probability to appear. The problem comes when we do not want a number in that interval [0, RAND_MAX], but in another interval, for example of size N. Then if we used rand()%N to get a number from 0 to N-1, the probability of each number appearing is only equal if N divides RAND_MAX+1, which is to say, very rarely.

With our function, what is intended is to solve this problem by using a parameter **t=(double)rand()/((double)RAND_MAX+1)**, which is a double in the range [0,1)- because the biggest t that we can get is RAND_MAX/(RAND_MAX+1) = 0.99999999953-, with equal probability among every possible value of t. Then we multiply this parameter by the size of the interval we want to find a random number +1; so, we make **t*(sup-inf+1)**. Now we have a number, that by construction is in the range [0, sup-inf+1) we only have to apply the floor function to it to get rid of the decimal part and have a number that goes from 0 to **sup-inf**. All possibilities are equally probable. Finally, we only add this result to **inf**, to get a number from **inf** to **sup** and return it.

This function is displayed in section 4.1.

6.2 Question 2

First, it orders the first two elements of the array, creating an ordered subarray of two elements. Then for every subarray of i ordered elements (until i==n) it takes the (i+1)th element and places it in its position. This is easier to do because the subarray of i elements has already been ordered in the previous step. Therefore, after this step we end up with an ordered subarray that is 1 element bigger than in the previous step. The algorithm ends when the subarray is of size N, size of the original table.

6.3 Question 3

Because the subarray of 1 element (the initial one) is already ordered; one-element arrays are always ordered. There is no need to compare it with any other element. Then the second element is compared to the first in order to create a subarray of two ordered elements.

6.4 Question 4

The basic operation is, as in most sorting algorithms, the key comparison. It is made (sometimes multiple times) when inserting an element into its corresponding place between its predecessors in the array.

6.5 Question 5

As discussed before with the different plots, the abstract running time obtained during the practice perfectly matches the theoretical result in the average case: we obtained a function that was $N^2/4+O(N)$.

For the maximum number of key comparisons, the function obtained was $0.4N^2+O(N)$. It differs a bit from the expect result, but it is still $\Theta(N^2/2)$.

Finally, the minimum number also differs as we showed before. The function obtained is Θ(N), in fact it was 2N+O(1). Again, the function obtained is Θ(N-1), the theoretical result.

The reasons of these deviations are explained in 5.5.1 and commented again in the conclusion (section 7).

6.5 Question 6

As shown by the graphs, both algorithms have the same behavior for maximum, minimum and average case. In fact, the code for both algorithms differs essentially just in one symbol “>” that is replace by “<”.

Therefore, it would make no sense if the algorithms had different asymptotical growths but worked in the same way.

7. Final Conclusions.

During this practice we have been able to compare theoretical results from class with data obtained during tests. As it happens most of the times, tests do not provide the exact same results as expected. This time the major differences were due to technical limitations; we could not put our computers to sort 10^8 permutations, nor could we measure time with as much precision as we wanted.

For the first issue, we had to accept the limitation of our equipment and order less permutations and take it into account when analyzing the results.

For the second one, we used high values of N so the algorithm took a high enough amount of time we were able to measure it; in this case our precision was of 10^{-2} seconds.