Analysis of Algorithms 2020/2021

Practice 1

Daniel Varela

Guillermo Martín-Coello Juárez

Group 1292

Code	Plots	Memory	Total

1. Introduction.

In this practice we worked on generating random permutations and then using different sort algorithms to check how fast they are. We used both InsertSort and InsertSortInv algorithms.

2. Objectives

Here you indicate the work you are going to do in each section.

2.1 Section 1

We need to create a function of type int that receives two integers indicating the minimum and the maximum numbers between which the function needs to generate a random number and return it.

2.2 Section 2

In this exercise we need to create a function that generates a permutation between N elements. It receives an integer indicating the amount of elements that the permutation must have and the function creates an array with elements from one to N and orders them in a random way using the function made in exercise 1. The function then returns a pointer to such permutation.

2.3 Section 3

This exercise generates an array of N integers from 1 to N and it makes n_perms permutations and it finally returns an array of arrays containing n_perms permutations of size N. It does so by calling the previous function generate_perm.

2.4 Section 4

This exercise is about implementing a function with the algorithm insert short to order elements from a table. The inputs are the table, the first and the last element to be ordered. the output is an error control, but also the given table will now be ordered.

2.5 Section 5

Exercise five checks the time it takes for a given algorithm to order different permutations of different sizes. The inputs are the order algorithm, a file to write the results, a minimum value of N being N the size of each perm, the maximum value of N, the increment of N and the number of perms for each N. the output is a control error and also it creates a file with all the measured times and results.

2.6 Section 6

In this exercise we just need to create an alternative version of the Insert Sort algorithm that orders on reverse the elements. The inputs are the table to be ordered, and the first and last elements to be ordered. The results are the same as the ones predicted by the theory. .The output is the ordered table.

3 Tools and Methodology

3.1 Section 1

Using Visual Studio in Linux, we made an algorithm that generates a random number from the minimum (inf) and maximum (sup). After that we used the tool GNUPlot to generate an histogram containing the possible numbers and the frequency in which they appear in a 10000 total numbers generated.

3.2 Section 2

Using Visual Studio in Linux, we made an algorithm that received an integer and generates a random permutation of such size. To do that, we wrote a loop that initializes the amount of values needed (N) in an array, after that, we moved the positions of such arrays using the function created in the last section. To test this, we modified the make so that it printed a random permutation of 5 integers 10 times.

3.3 Section 3

Using Visual Studio in Linux, in this section, we created a function that allows us to create an array of different permutations, using a loop and the function created in the last section. To test the result, we changed the makefile so it gave us 10 permutations of a size 7 array.

3.4 Section 4

Using Visual Studio in Linux, we created an InsertSort algorithm that receives a permutation and two integers which indicate its first and last term. The algorithm goes through the first two components of the permutation and compares them, switching them if the left one is bigger than the right one, then it does the same thing but for the first three elements and so on. In order to try this section, we used the makefile of exercise 4 so that it generated a permutation of 10 numbers and then used InsertSort to order the permutation from the smallest number to the largest.

3.5 Section 5

Using Visual Studio in Linux, we made three functions:

One of which (average_sorting_time) calculated the average time, the minimum basic operation, the maximum and the average and assigned them to each corresponding site in the structure. To do so we made two variables that contained to which we assigned the current time and initialized them at different points of the program so one of them stored the time in which the program starts to order the permutations and the other one stored the final time, with those variables we were then able to calculate the average time per permutation.

The second function (generate_sorting times) is the main one, the one that calls the other two.

The third function (generate_sorting times) gets the size, the average time, the average number of basic operations, the minimum and the maximum from the struct and prints the values in a .log file.

3.6 Section 6

Using Visual Studio in Linux, we created a variation from InsertSort called InsertSortInv which did the same thing but ordered the values in inverse order (from largest to smallest).

4. Source code

4.1 Section 1

```

/**************************************************/
/* Function: random_num           Date:   05/10/20   */
/* Authors: Guillermo Martin-coello & Daniel Varela*/
/*                                                */
/* 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)
{
  int result;
  if (inf == ERR || sup == ERR)
  {
    return ERR;
  }
  result = (rand() % (sup - inf + 1)) + inf;
  return result;
}
```

4.2 Section 2

```c
/**********************************************************/
/* Function: generate_perm         Date: 05/10/20    */
/* Authors: Guillermo Martin-coello & Daniel Varela*/
/*                                                   */
/* Routine that generates a random permutation        */
/*                                                   */
/* Input:                                            */
/* int N: number of elements in the permutation      */
/* Output:                                           */
/* int *: pointer to integer array                    */
/* that contains the permutation                      */
/* or NULL in case of error                           */
/**********************************************************/
int *generate_perm(int N)
{
  int i, aux, j;
  int *perm;
  if (N<=0)
  {
    return NULL;
  }
  perm = malloc(N*sizeof(perm[0]));
  if (perm == NULL)
    return NULL;
  for (i = 0; i < N; i++)
  {
    perm[i] = i + 1;
  }
  for (i = 0; i < N; i++)
  {
    j = random_num(i, N - 1); /*random num*/
    aux = perm[i];            /*auxiliar num*/
    perm[i] = perm[j];
    perm[j] = aux; /*perm*/
  }
  return perm;
}
```

4.3 Section 3

```
/***********************************************************/
/* Function: generate_permutations   Date: 05/10/20*/
/* Authors: Guillermo Martin-coello & Daniel Varela*/
/*                                                 */
/* 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 i, **array;
  if(n_perms<0 || N <=0){
    return NULL;
  }
  array = (int **)calloc(n_perms, sizeof(array[0]));
  /*tenemos que cambiar el calloc por malloc*/
  if (array == NULL)
  {
    return NULL;
  }
  for (i = 0; i < n_perms; i++)
  {
    array[i] = generate_perm(N);
    if (array[i] == NULL)
    {
      i--;
      while (i >= 0)
      {
        free(array[i]);
        i--;
      }
      return NULL;
    }
  }
  return array;
}
```

4.4 Section 4

```

/***********************************************/
/* Function: InsertSort            Date: 18/10/20  */
/* Authors: Guillermo Martin-coello & Daniel Varela*/
/*                                                 */
/* This function orders the table given by the input*/
/* table from the smallest number to the biggest    */
/* using insertsort algorithm                       */
/*                                                 */
  /* Input:                                          */
  /* int table: array with all the values            */
  /* int ip: first element of the table              */
  /* int iu: last element of the table               */
  /* Output:                                         */
  /* int: It returns an integer containig tne amount */
  /* of basic operations realized and ERR (-1) if    */
  /* something goes wrong                            */
  /*************************************************/
  int InsertSort(int *table, int ip, int iu)
  {
    int aux, i, j, ob = 0;
    if (table == NULL || ip == -1 || iu == -1)
    {
      return ERR;
    }
    for (i = ip + 1; i <= iu; i++)
    {
      aux = table[i];

      j = i - 1;
      if (table[j] <= aux)
        ob++;
      while (j >= ip && ob++ && table[j] > aux)
      {
        table[j + 1] = table[j];
        j--;
      }
      table[j + 1] = aux;
    }
    return ob;
  }
```

4.5 Section 5

```c
/**********************************************/
/* Function: average_sorting_time  Date: 15/10/20  */
/* Authors: Guillermo Martin-coello & Daniel Varela*/
/*                                               */
/* This function adds the corresponding fields to  */
/* the structure ptime, adding its number of       */
/* permutations, the size of the permutations, the */
/* average execution time, the average number of   */
/* times OB was executed alongside with the minimum */
/* amount of times and the maximum                  */
/*                                                  */
/* Input:                                           */
/* pfunc_ordena metodo: function of the method we   */
/* want to use                                      */
/* int n_perms: Number of permutations to be made   */
/* int N: number of elements in every permutation   */
/* Output:                                          */
/* short: It returns ON (0) if everything goes as   */
/* planned and ERR (-1) if there has been an error */
/**********************************************/
short average_sorting_time(pfunc_ordena metodo, int n_perms, int N, PTIME_AA ptime)
{

  int i = 0;
  clock_t startClock;
  clock_t endClock;
  double t;
  int ob;
  int min_ob = INT_MAX, max_ob = 0;
  double average_ob = 0,dob;
  int **perm=NULL;
  ptime->N = N;
  ptime->n_elems = n_perms;
  if (metodo == NULL || n_perms < 0 || N < 0 || ptime == NULL)
  {
    return ERR;
  }
  perm = generate_permutations(n_perms, N);
  if (perm == NULL)
  {
    return ERR;
  }
  startClock=clock();
  if(startClock<=0){
    return ERR;
  }
  for (i = 0; i < n_perms; i++)
  {
    ob = metodo(perm[i], 0, N-1);
    if (ob == ERR)
    {
```

```
      for (i=i-1; i >= 0; i--)
      {
        free(perm[i]);
      }
      free(perm);
      return ERR;
    }
    if (ob < min_ob)
    {
      min_ob = ob;
    }
    if (ob > max_ob)
    {
      max_ob = ob;
    }
    dob = (double)ob;
    average_ob = average_ob + dob;
  }
  endClock=clock();
  if(endClock<=0){
    return ERR;
  }
  if(n_perms!=0){
  average_ob = (average_ob / (double)n_perms);
  }


  for (i = 0; i < n_perms; i++)
  {
    free(perm[i]);
  }
  free(perm);
  ptime->max_ob = max_ob;
  ptime->min_ob = min_ob;
  ptime->average_ob = average_ob;
  t = (double)((endClock - startClock) /n_perms)/CLOCKS_PER_SEC;
  ptime->time = t;
  return OK;
}


/*****************************************************/
/* Function: generate_sorting_times                  */
/* Date: 17/10/20                                    */
/* Authors: Guillermo Martin-coello & Daniel Varela*/
/*                                                   */
/* This function generates a table of permutations */
/* and then it calls the average sorting time       */
/* function so it adds its corresponding values to */
/* its corresponding fields. Afterwards it makes a */
/* file where it stores the the average clock time,*/
/* and the average, minimum and maximum times that */
/* OB was called using the method method            */
/*                                                   */
/* Input:                                            */
```

```c
/* pfunc_ordena metodo: function of the method we   */
/* want to use                                      */
/* int n_perms: Number of permutations to be made   */
/* int num_min: minimun number of elements in the   */
/* permutations                                     */
/* int num_max: maximun number of elements in the   */
/* permutations                                     */
/* incr: increment value of elements in the         */
/* permutations                                     */
/*                                                  */
/* Output:                                          */
/* short: It returns ON (0) if everything goes as   */
/* planned and ERR (-1) if there has been an error */
/***************************************************/
short generate_sorting_times(pfunc_ordena method, char *file, int num_min, int num_max, int incr, int n_perms)
{
  int n,i;
  PTIME_AA ptime = NULL;
  int n_times = ((num_max - num_min) / incr) + 1;
    if (method == NULL || n_perms < 0 || num_min < 0 || num_max < 0 || incr < 0 )
  {
    return ERR;
  }


  ptime = (TIME_AA*)malloc(n_times*sizeof(ptime[0]));
  if(ptime == NULL)
  {
    return ERR;
  }

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

  return 0;
}
```

```c
/***************************************************/
/* Function: save_time_table Date:                 */
/* Date: 17/10/20                                  */
/* Authors: Guillermo Martin-coello & Daniel Varela*/
/*                                                 */
/* This function generates a file file where it    */
/* stores the size, the average clock time and     */
/* and minimum times OB is executed for each ptimme*/
/*                                                 */
/* Input:                                          */
/* char *file: File to be written in               */
/* PTIME_AA ptime: variable to store time data     */
/* int ntimes: number of times the calculations    */
/* are done                                        */
/***************************************************/
short save_time_table(char *file, PTIME_AA ptime, int n_times)
{
  FILE *f;
  int i=0;
  if((f=fopen(file, "w"))== NULL)
  {
    return ERR;
  }
  for(i=0; i<n_times; i++) {

    if(fprintf(f, "%d  %.3f  %f  %d  %d \n", ptime[i].N, ptime[i].time*1000, ptime[i].average_ob, ptime[i].max_ob, ptime[i].min_ob) == ERR)
    {
      return ERR;
    }
  }

  if(fclose(f) == ERR)
  {
    return ERR;
  }
  return 0;


}
```

4.6 Section 6

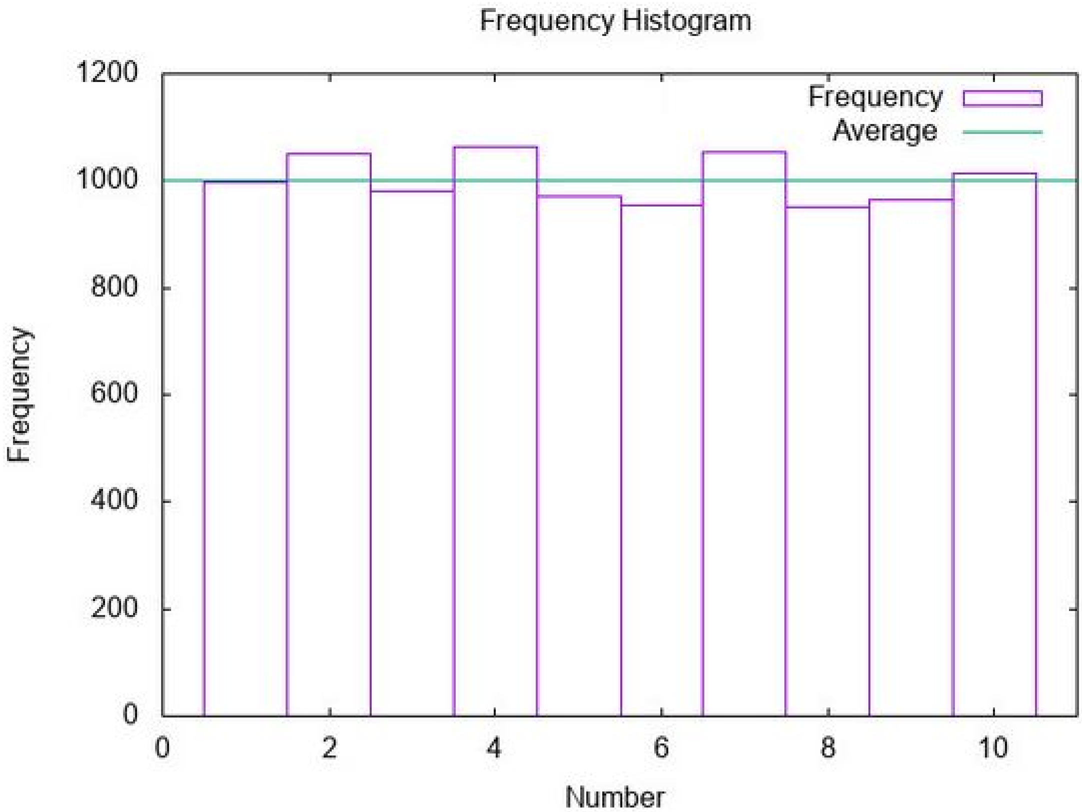
```c

/*************************************************/
/* Function: InsertSortInv         Date: 18/10/20  */
/* Authors: Guillermo Martin-coello & Daniel Varela*/
/*                                                 */
/* This function orders the table given by the input*/
/* table from the largest number to the smallest    */
/* using insertsort algorithm                       */
/*                                                 */
/* Input:                                           */
/* int table: array with all the values             */
/* int ip: first element of the table               */
/* int iu: last element of the table                */
/* Output:                                          */
/* int: It returns an integer containig tne amount */
/* of basic operations realized and ERR (-1) if     */
/* something goes wrong                             */
/*************************************************/
int InsertSortInv(int *table, int ip, int iu)
{
  int aux, i, j, ob = 0;
  if (table == NULL || ip == -1 || iu == -1)
  {
    return ERR;
  }
  for (i = ip + 1; i <= iu; i++)
  {
    aux = table[i];
    j = i - 1;
    while (j >= ip && ob++ && table[j] < aux)
    {
      table[j + 1] = table[j];
      j--;
    }
    table[j + 1] = aux;
  }
  return ob;
}

```

5. Results, Plots

Here you write the results obtained in each section, including the required plots.

5.1 Section 1

We first used the test to make a 15 value sample that generates random numbers from 1 to 5 to see if the random number generator was working properly, after checking that we made a 10000 values sample of random numbers ranging from 1 to 10 and printed them in a ".txt" file. From that ".txt" file we generated a histogram using GNUPlot and this was the result:



As we can see the plot shows us that our random number generator is working since each number appears more or less 1000 times, but it's not perfect, there exist lots of more efficient ways to generate random numbers. We will find another way to generate a random number in question 1.

5.2 Section 2

```
eps@labvirteps:~/Escritorio/AALG$ make exercise2_test
Running exercise2
Practice number 1, section 2
Done by: Daniel Varela & Guillermo Martín-Coello
Group: T05
5 2 4 1 3
4 5 1 2 3
3 5 2 4 1
5 1 3 2 4
2 3 1 4 5
1 3 2 5 4
5 3 2 4 1
3 5 1 2 4
5 1 4 3 2
1 2 5 3 4
```

The result is the one expected, the exercise 2 test returns 10 random permutations of size 5.

5.3 Section 3

```
eps@labvirteps:~/Escritorio/AALG$ make exercise3_test
Running exercise3
Practice number 1, section 3
Done by: Daniel Varela & Guillermo Martín-Coello
Group: T05
7 2 5 4 1 6 3
5 1 2 4 7 6 3
4 3 5 6 1 7 2
7 4 6 3 5 2 1
7 3 1 2 4 5 6
6 3 2 4 7 5 1
2 7 4 6 1 5 3
6 7 3 4 2 1 5
1 3 7 2 5 6 4
7 1 6 3 4 5 2
```

We can see here the result of exercise 3, in this case we generated 10 random permutations of size 7.

5.4 Section 4

```
eps@labvirteps:~/Escritorio/AALG$ make exercise4_test
Running exercise4
Practice number 1, section 4
Done by: Daniel Varela & Guillermo Martín-Coello
Group: T05
1       2       3       4       5       6       7
```

This test gives the resulting permutation of a random permutation of size 7 after applying InsertSort on it. As we can see it's ordered from the smallest number (1) to the largest (7).

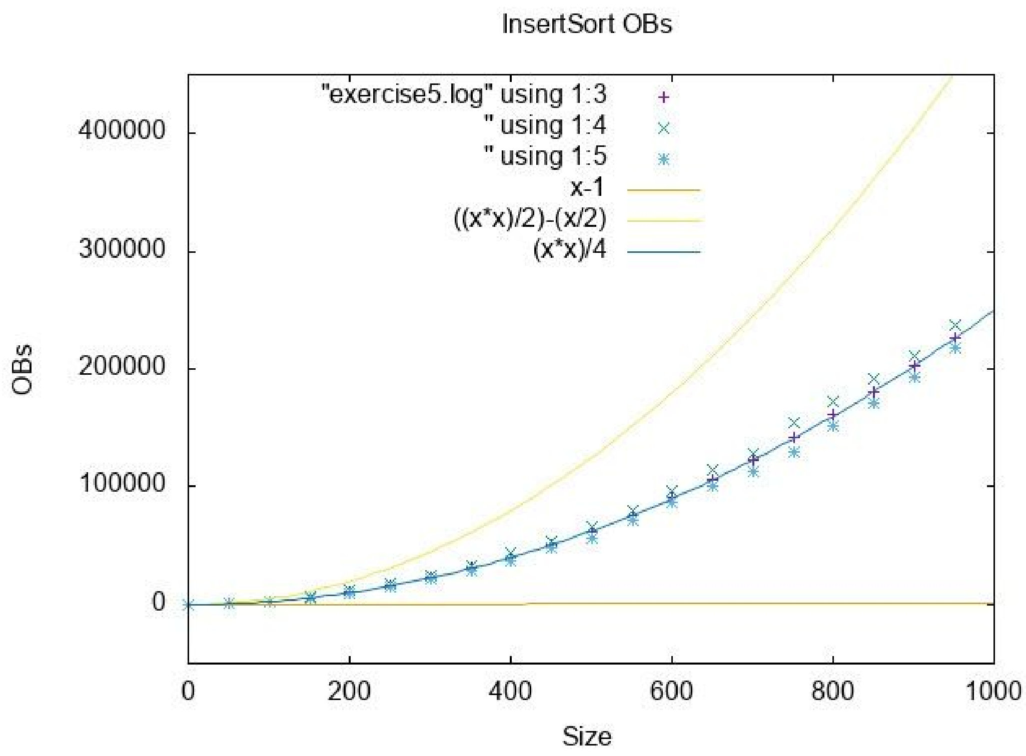
5.5 Section 5

After executing exercise 5 we get a file that shows the size, the average time and the worst, average and best case of ob. It looks just like this:

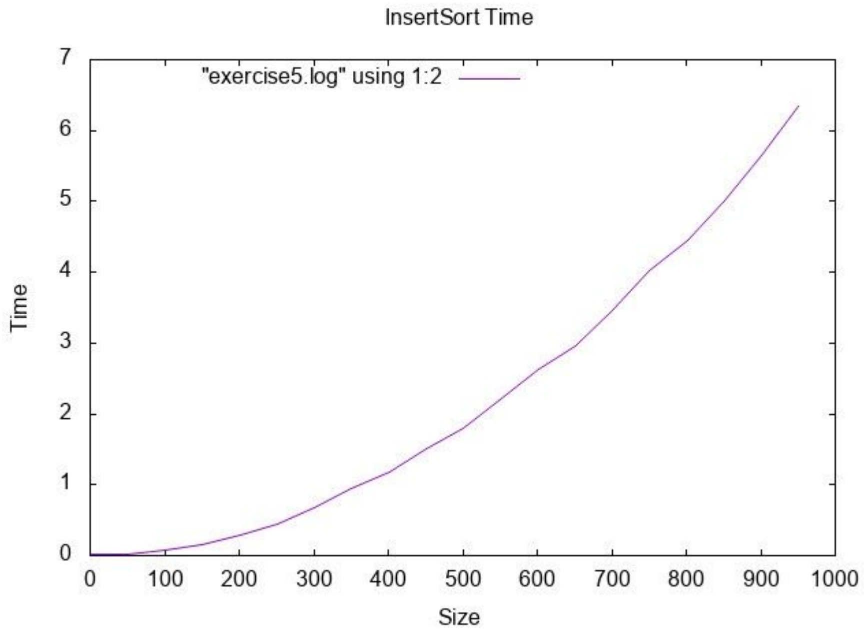
```
1000  0.590  249798.500000  263381  240942
2000  2.358  996023.700000  1007996  977203
3000  6.212  2275847.200000  2318868  2256004
4000  11.498  4004699.400000  4101044  3953999
5000  17.870  6288097.300000  6391068  6180111
6000  25.490  9042765.900000  9140781  8857987
7000  34.791  12249864.900000  12417959  12124625
8000  44.893  15970161.300000  16051718  15829973
9000  57.726  20285990.200000  20450279  19899439
10000  71.978  25012272.200000  25198182  24806447
```

With this generated file we made some plots to represent the data, the first one shows the worst, average and best case of ob and to do so more clearly we also included the theoretical best, worst and average case.

This is the plot:

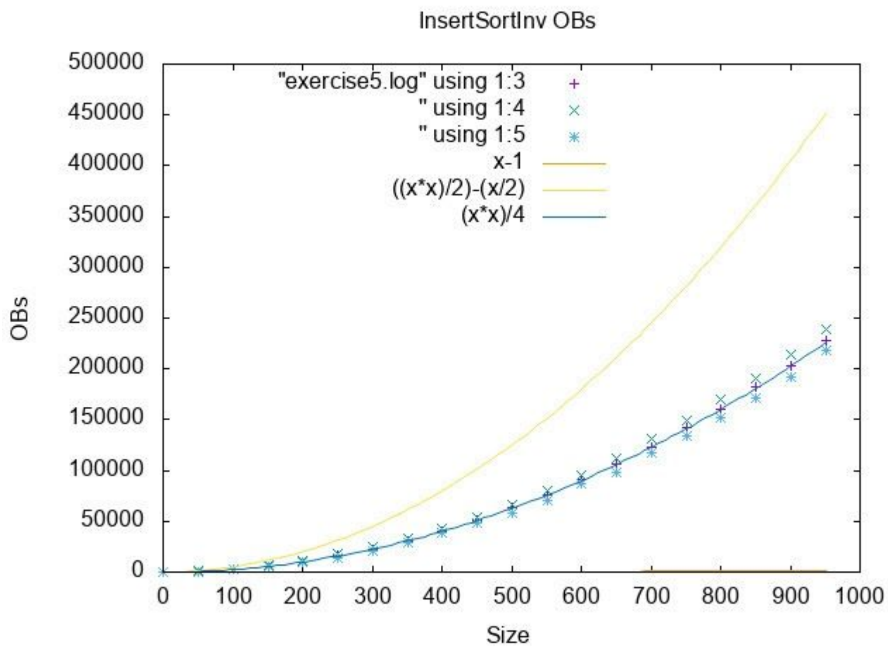


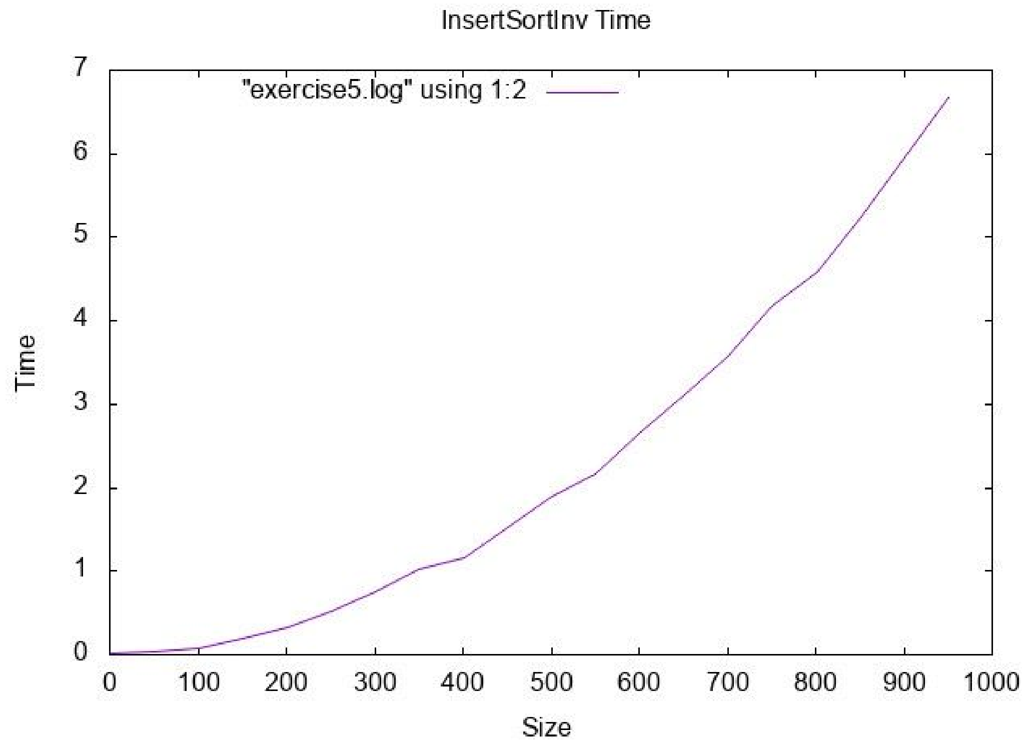
After this, we made another plot to represent the average sorting time:



5.6 Section 6

We repeated the same procedure as the one mentioned in section 5 and obtained the following plots:





5. Answers to theoretical Questions.

5.1 Question 1:

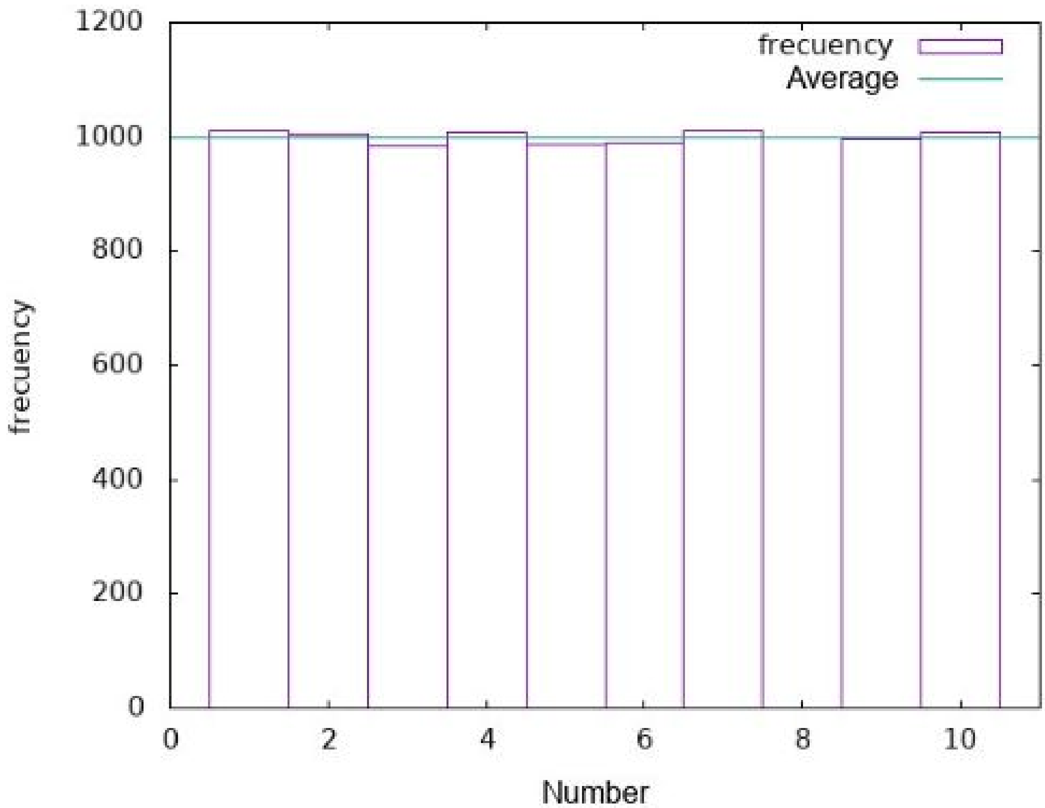
Justify your implementation of ***random_num****. In what ideas is it based? What book/article, if any, have you taken the idea? Propose an alternative method of generating random numbers and Justify your advantages/disadvantages regarding your choice.*

The random_num function is based on the ideas of Park and Miller, wroten in the book "Numerical Recipes in C, the art of Scientific computing".
Bibliography:

William H. Press, Saul A. Teutolsky, William T. Vertteling, Brian P. Flannery. (1988). Numerical Recipes in C, the art of Scientific computing. Cambridge: American Institute of Physics.

We tried to implement this code to improve the random results.

```c
#define IA 16807
#define IM 2147483647
#define AM (1.0 / IM)
#define IQ 127773
#define IR 2836
#define MASK 123459876

float ran0(long *idum)
{
  long k;
  float ans;
  *idum ^= MASK;
  k = (*idum) / IQ;
  *idum = IA * (*idum - k * IQ) - IR * k;
  if (*idum < 0)
    *idum += IM;
  ans = AM * (*idum);
  *idum ^= MASK;
  return ans;
}

int random_num(int inf, int sup)
{
int rand(void);
 int num;
 float multiplyer;
 long randomgenerator = rand();
 multiplyer = ran0(&randomgenerator);
  multiplyer = (100 * multiplyer);
  num =(int)multiplyer;
  num = (num % (sup - inf + 1)) + inf;
 return num;
}
```



5.2 Question 2

*Justify as formally as you can the correction (or put another way, why order well) of the algorithm **InsertSort**.*

The algorithm insert sort sorts all elements on an array. For doing that it runs from the second element to the last one,
and for each of them it checks if the previous one is higher than it. If so, it changes the position of both elements an
then checks again with the previous one to repeat the cycle.

5.3 Question 3

*Why does the outer loop of **InsertSort** not act on the first element of the table?*

The outer loop of InsertSort doesn't act on the first element of the table because it doest have any previous element to compare with in the array hence it's already ordered because we are just ordering one number.

5.4 Question 4

*What is the basic operation of **InsertSort**?*

The basic operation of InsertSort is `table[j] > aux` because it's the operation that's most called in the function since the code that is inside the while is depending on a condition, hence it can not be the basic operation but the condition is.

5.5 Question 5

*Give execution times based on the input size n for the worst case WBS(n) and the best case BBS(n) of **InsertSort**. Use asymptotic notation (O, Θ, or, Ω, etc.) whenever possible.*

For the worst case the InsertSort algorithm has an execution time of N^2 because
$W(N)=\max \{n\ A\ (I)/\ I \in S\ A\ (N)\}$
The worst case would be $W = \Omega(n^2/2-n/2)$.
and for the best case it has an execution time of N because
$B\ (N)=\min \{n\ (I)/\ I \in S\ A\ (N)\}$
The best case would be $B = O(n-1)$.

5.5 Question 6

*Compare the times obtained for **InsertSort** and **InsertSortInv**, justify the similarities or differences between the two (that is, indicate if the graphs are the same or different and why).*

As shown by the graphs we obtained some pretty similar times, this is because they are the same algorithm and the only thing that changes is the order from lowest to highest to the other way around, the only differences we might encounter are for some specific cases, since for example one's best case is the other's worst case. Nevertheless the average time they get is almost the same since the permutations are made at random. They both grow exponentially.

6. Final Conclusions.

With this practice we learned how to create a series of random permutations and measure the execution time of a sorting function . This is very useful in order to make a more efficient code and learn about how a sorting algorithm works. We also learned how to make histograms using the tool GNUPlot which allows us to visualize data that we retrieve from programas. In a nutshell we started to learn how to take the basics of algorithm analysis into practice using C.