Analysis of Algorithms 2020/2021

Practice 1

Sofia Xiao Fernandez Marin, 1292.

Formato de entrega/nombre fichero -1
Valgrind didn't pass on exercise4 and exercise5 -> **Final grade 3.5**

Code	Plots	Memory	Total
2.65	1.1	1.25	3.5

1. Introduction

In this practice we have to write code and implement several functions to generate permutations of numbers and later use that permutations to prove sorting algorithms functions.

2. Objectives

-0.1

2.1 Section 1

Actually, we only had 3 sections with subsections

In this part, we have learnt how to use the rand and srand function to create a routine that generates equiprobable random numbers between the values we provide to the routine. In our case, we have chosen to generate just positive numbers, not negative ones.

2.2 Section 2

In this section we have implemented a function able to create random permutations based on a given table.

2.3 Section 3

After creating the function that generates random permutations, we need to code another function that, calling the previous one, generates a matrix of equiprobable permutations.

To improve understanding, you could mention the definition of the functions to implement and bold or underline them.

2.4 Section 4

In this part we implemented the function InsertSort, that sorts the elements of a given table of random numbers. It returns the number of times the basic operation has been executed.

2.5 Section 5

In this section we have implemented three different functions to measure the execution time and the basic operations of the sorting algorithm InsertSort.

2.6 Section 6

Finally, we have created the function InsertSortInv, to sort a table in a descendant way, and to compare the results between this function and the InsertSort function.

3 Tools and Methodology -0.2

Basically, the tool that we have been using during this task is Visual Studio Code, because it facilitates us the debugging of the code and, we could program at the same time and the same code with different screens. We have also executed all the exercises with valgrind to prove that there were no dynamic memory allocation errors.

Text justified

3.1 Section 1

In this first exercise we looked for information about the rand and srand functions to know better their functionality, and then we fit the values in rand to generate equiprobable random numbers between the values given to the random_num function.

Here it was expected a more detailed explanation of how you implemented the random function: Discussions, bibliography, internet, tests, etc.

3.2 Section 2

To do this exercise, that consisted in implementing the generate_perm function, we just transformed the given pseudocode into C code, and then we proved with valgrind that memory allocation was in order.

3.3 Section 3

In this case, we have to create a matrix of permutations. So, we reserved memory for the double pointer and then we initialized each line of the matrix calling the previous function generate_perm. We also checked memory errors with valgrind.

3.4 Section 4

Here we implemented the function InsertSort. The teacher helped us with the pseudocode in class, so it was not so difficult. The code we implemented starts the loop in the second element of the table, so that a number doesn‘t swap with each self.

3.5 Section 5

In this section, we first implemented the function average_sorting_time, that first creates a table of permutations, then it sorts each line of the table through a loop that goes through the lines and call the method for each of them. We also used the function clock to establish the time it takes to sort. At the end of the program we free the table to avoid errors.

Then, we implemented the function generate_sorting_times. Where first we obtained the number of permutations we are going to work on. Then we reserved memory for a structure of time for each permutation to save the executing information about each permutation. Then we generated the permutations and saved them in the corresponding structures. Finally, we called the function save_time_table to write in a file all the information of each permutation.

3.6 Section 6

In this last section, we implemented the function InsertSortInv in the sorting.c file. This function was implemented similar as in InsertSort but sorting a list of numbers in reverse order.

4. Source code

4.1 Section 1

```
int random_num(int inf, int sup){

  int random = 0;

  if (sup < inf|| inf < 0|| sup > RAND_MAX) return ERR;

  random = (int) inf + rand() / ((RAND_MAX + 1.0) / (sup - inf + 1));

  return random;
}
```

4.2 Section 2

-0.3

```
int* generate_perm(int N){

  int i = 0, num1 = 0, num2 = 0;
  int *perm = NULL;

  if (N <= 0) return NULL;

  perm = (int*) malloc ( N * sizeof(int));
```

sizeof(perm[0])
In the delivered code you have a calloc. Why codes doesn't match?

```
  if (perm == NULL) return NULL;

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

perm[i] = i + 1; In order to have values from 1 to N

```
  }

  for (i = 0; i < N; i++){
    num1 = perm[i];
    num2 = random_num(i, N-1);
    if (num2 == -1){
      free (perm);
      return NULL;
    }
```

This can't happen in this case. Unnecessary code

```
    perm[i] = perm[num2];
    perm[num2] = num1;
  }

  return perm;
}
```

4.3 Section 3

-1.25

```
int** generate_permutations(int n_perms, int N){

  int **perm = NULL;
  int i=0;

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

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

  for (i = 0; i < n_perms; i++){
      perm [i]= generate_perm(N);

      if (perm[i] == NULL) {
          for (i = i-1; i<= 0; i--) {
            free (perm[i]);
          }
          free(perm);
          return NULL;
      }

  }

  return perm;
}
```

Better (perm == NULL)

while i >=0 This looks like an infinitive loop...

4.4 Section 4

```
int InsertSort(int* table, int ip, int iu){

  int i = 0, j = 0, aux = 0, cont = 0;

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

  for (i = ip + 1; i <= iu; i++){
    aux = table[i];
    j = i - 1;
    while (j >= ip && table[j] > aux){
      table[j + 1] = table[j];
      j--;
      cont++;
    }

    if (table[j]<=aux) cont++;

    table[j + 1] = aux;
  }

  return cont;
}
```

This could be improved a bit

4.5 Section 5 -2.2

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

  clock_t time1 = 0, time2 = 0;
  int i = 0, j = 0, time_ob = 0, ob=0;
  int **table = NULL;
  double times = 0;

  ptime->average_ob = 0;
  ptime->max_ob = 0;
  ptime->min_ob = 0;   It would have been better to initialize the minimum with a very high value
  ptime->N = 0;        as recommended
 ptime->n_elems = 0;
  ptime->time = 0;

  if(!method || n_perms<=0 || N<=0 || !ptime) return ERR;

  table = generate_permutations (n_perms, N);
  if (!table) return ERR;

  time1 = clock();   Missing error handling of clock
  for (i = 0; i < n_perms; i ++){
    time_ob = method(table[i], 0, N - 1);
    if (time_ob == ERR){
        for(j = i; (j = 0); j--) {
          free(table[j]);
          free(table);   This needs to be outside the for, otherwise you're loosing memory
          return ERR;
        }
    }
    if (ptime->min_ob == 0 || ptime->min_ob > time_ob) ptime->min_ob = time_ob;
    if (ptime->max_ob == 0 || ptime->max_ob < time_ob) ptime->max_ob = time_ob;
    ob+=time_ob;
  }

  time2 = clock();  Missing error handling of clock
  times += ((time2-time1)/(double) CLOCKS_PER_SEC);   You could have changed it to ms or ns
  times = times/n_perms;   Missing casting (double) n_perms

  ptime->average_ob = (double) ob/n_perms;

  ptime->N = N;
  ptime->n_elems = i;   i < n_perms... Why didn't you put just n_perms?
  ptime->time = times;

  for(i=0;i<n_perms;i++) free(table[i]);
  free(table);

  return OK;
}
```

```
short generate_sorting_times(pfunc_sort method, char* file, int num_min, int
num_max, int incr, int n_perms) {

  int i = 0, num = 0, counter = 0, control = 0;
  PTIME_AA ptime = NULL;

  if (method == NULL || file == NULL || num_min <0 || num_min > num_max || incr <=
0 || n_perms <= 0) return ERR;


  /*Number of permutations*/
  num = (num_max - num_min) / incr + 1;

  ptime = malloc(num * sizeof(TIME_AA));

  if (ptime == NULL) return ERR;

  /*Generates permutations and shave them in the corresponponding structures */
  for (i = num_min; i<= num_max; i = i + incr, counter++) {
    control = average_sorting_time(method, n_perms, i, &ptime[counter]);
      if (control == ERR) {
        free(ptime);
        return ERR;
      }
  }
  /*Call the save_time_table function to print all data obtained*/

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

  free(ptime);

  return OK;
}
```

```
short save_time_table(char* file, PTIME_AA ptime, int n_times){
```
-0.1

```
  int i=0;
  FILE *f = NULL;

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

  f = fopen (file, "w");

  if (f== NULL) return ERR;

  fprintf(f, "N\t");
  fprintf(f, "time\t");
  fprintf(f, "average ob\t");
  fprintf(f, "max ob\t");
  fprintf(f, "min ob\t");
  fprintf(f, "\n");
```

You could just keep ypur file clean by avoiding printed texts. Just have ready-to-use data on it.

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

fprintf could fail

```
fclose(f);
```

Missing error handling for fclose

```
return OK;
}
```

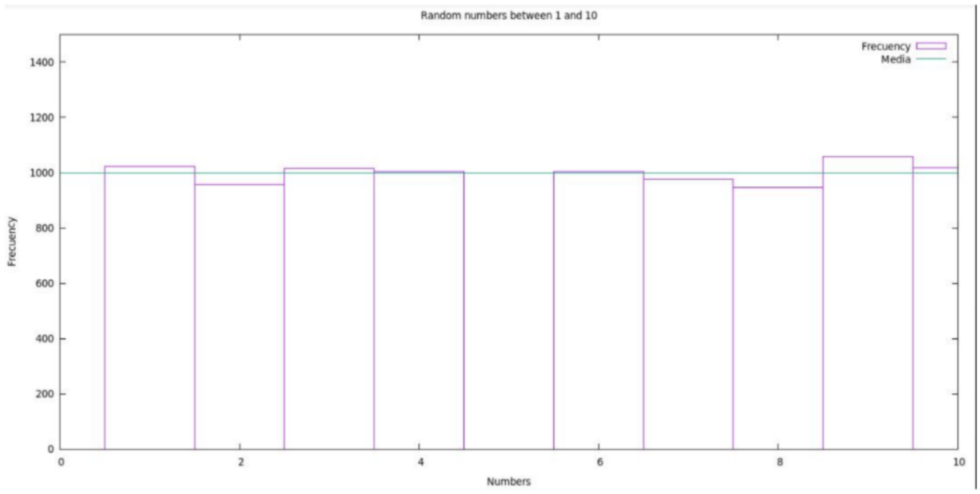
4.6 Section 6

```
int InsertSortInv(int* table, int ip, int iu){

  int i = 0, j = 0, aux = 0, cont = 0;

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

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

5. Results, Plots

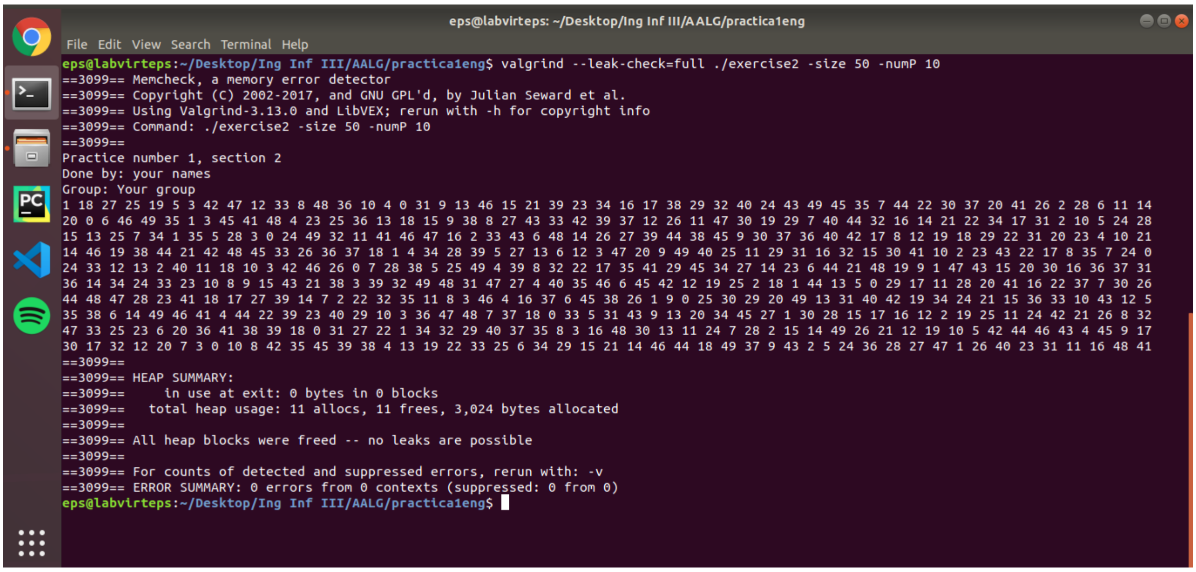
5.1 Section 1

Nice but sample is too small! This is not representative

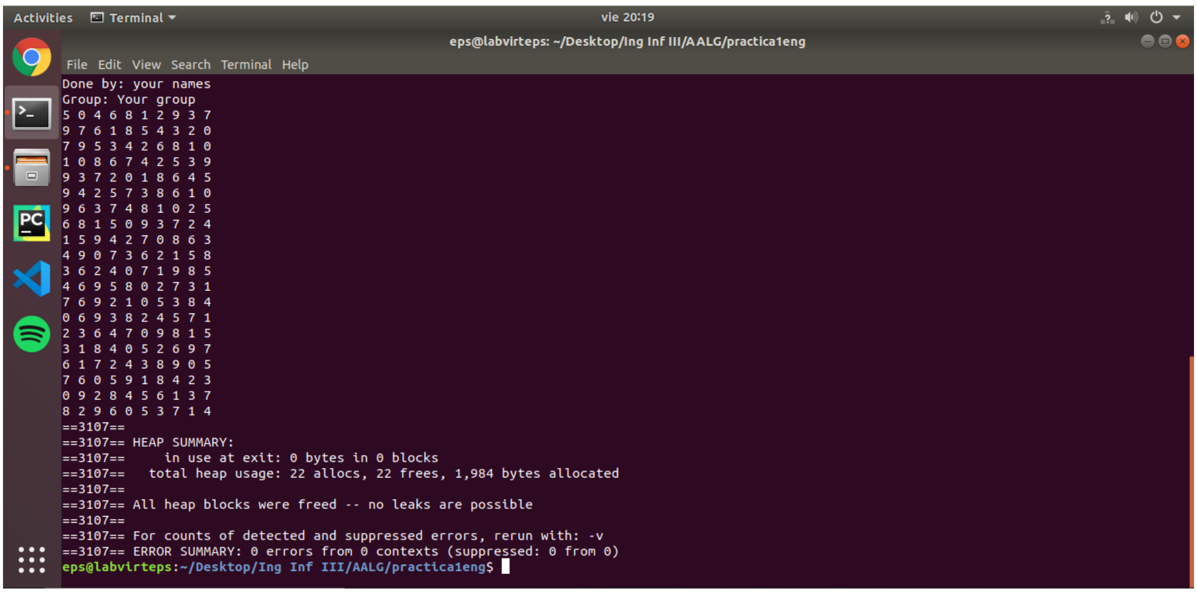


As we can see, there is a straight line because the numbers appear the same time.

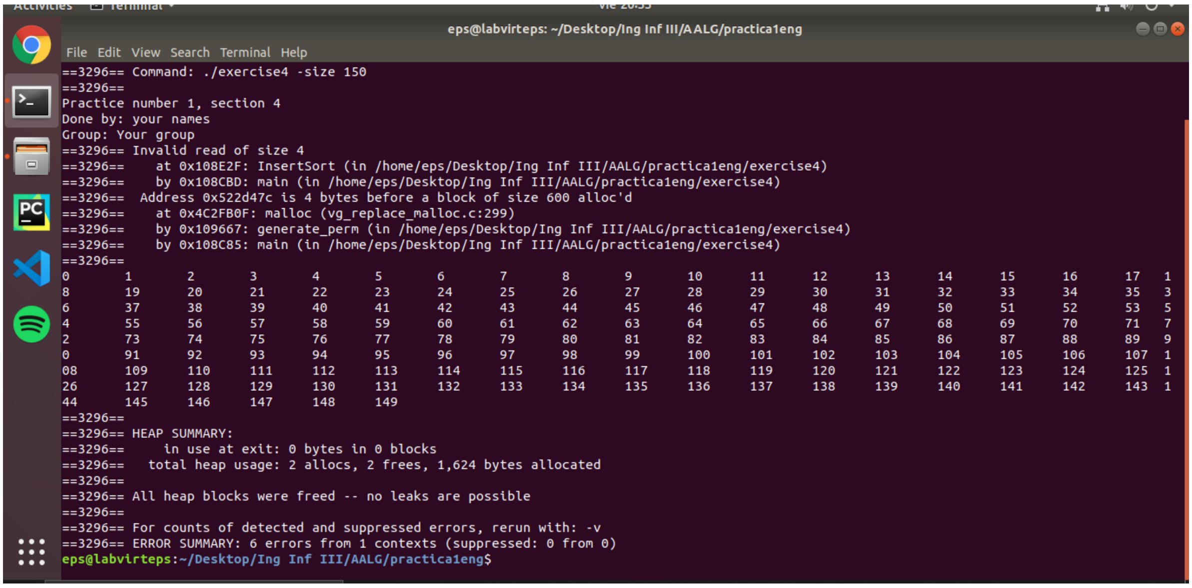
5.2 Section 2



5.3 Section 3



5.4 Section 4

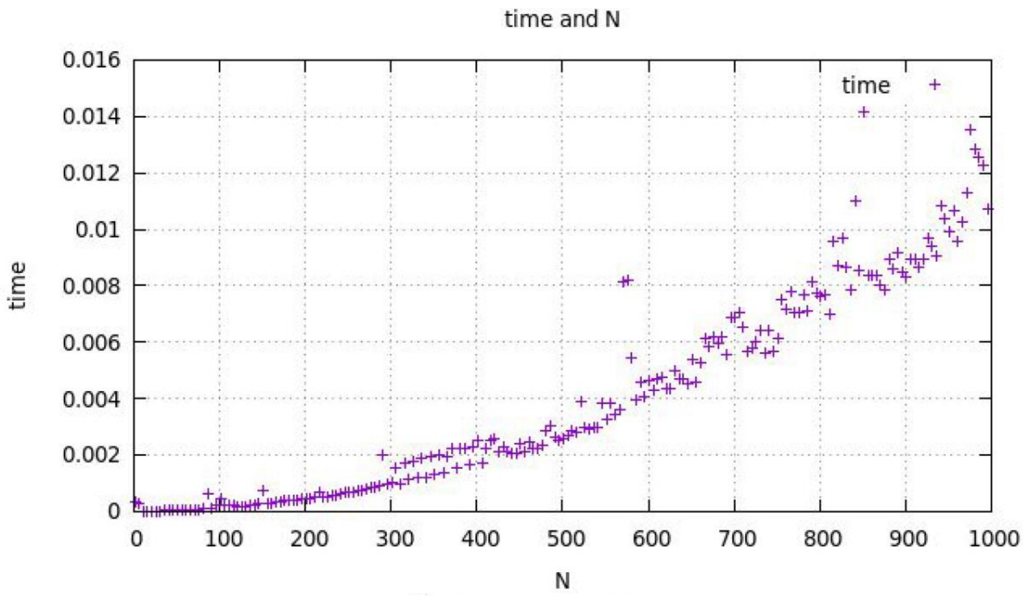


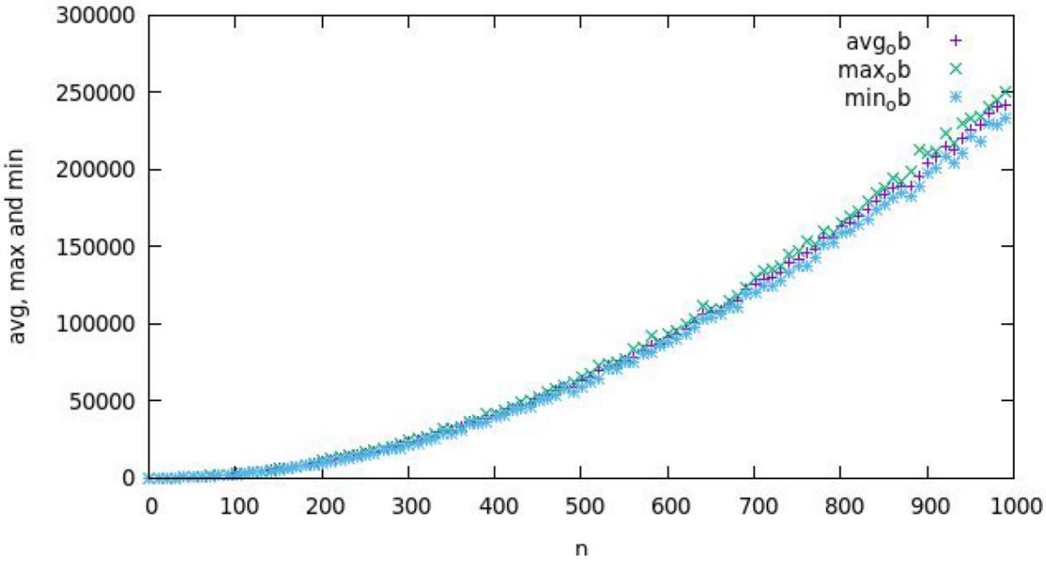
eps@labvirteps: ~/Desktop/Ing Inf III/AALG/practica1eng
File Edit View Search Terminal Help
==3296== Command: ./exercise4 -size 150
==3296==
Practice number 1, section 4
Done by: your names
Group: Your group
==3296== Invalid read of size 4
==3296==   at 0x108E2F: InsertSort (in /home/eps/Desktop/Ing Inf III/AALG/practica1eng/exercise4)
==3296==   by 0x108CBD: main (in /home/eps/Desktop/Ing Inf III/AALG/practica1eng/exercise4)
==3296== Address 0x522d47c is 4 bytes before a block of size 600 alloc'd
==3296==   at 0x4C2FB0F: malloc (vg_replace_malloc.c:299)
==3296==   by 0x109667: generate_perm (in /home/eps/Desktop/Ing Inf III/AALG/practica1eng/exercise4)
==3296==   by 0x108C85: main (in /home/eps/Desktop/Ing Inf III/AALG/practica1eng/exercise4)
==3296==
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 1
8 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 3
6 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 5
4 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 7
2 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 9
0 91 92 93 94 95 96 97 98 99 100 101 102 103 104 105 106 107 1
08 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 1
26 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 1
44 145 146 147 148 149
==3296==
==3296== HEAP SUMMARY:
==3296==   in use at exit: 0 bytes in 0 blocks
==3296==   total heap usage: 2 allocs, 2 frees, 1,624 bytes allocated
==3296==
==3296== All heap blocks were freed -- no leaks are possible
==3296==
==3296== For counts of detected and suppressed errors, rerun with: -v
==3296== ERROR SUMMARY: 6 errors from 1 contexts (suppressed: 0 from 0)
eps@labvirteps:~/Desktop/Ing Inf III/AALG/practica1eng$

-0.1

5.5 Section 5

This plot could look better if you use 'with lines'





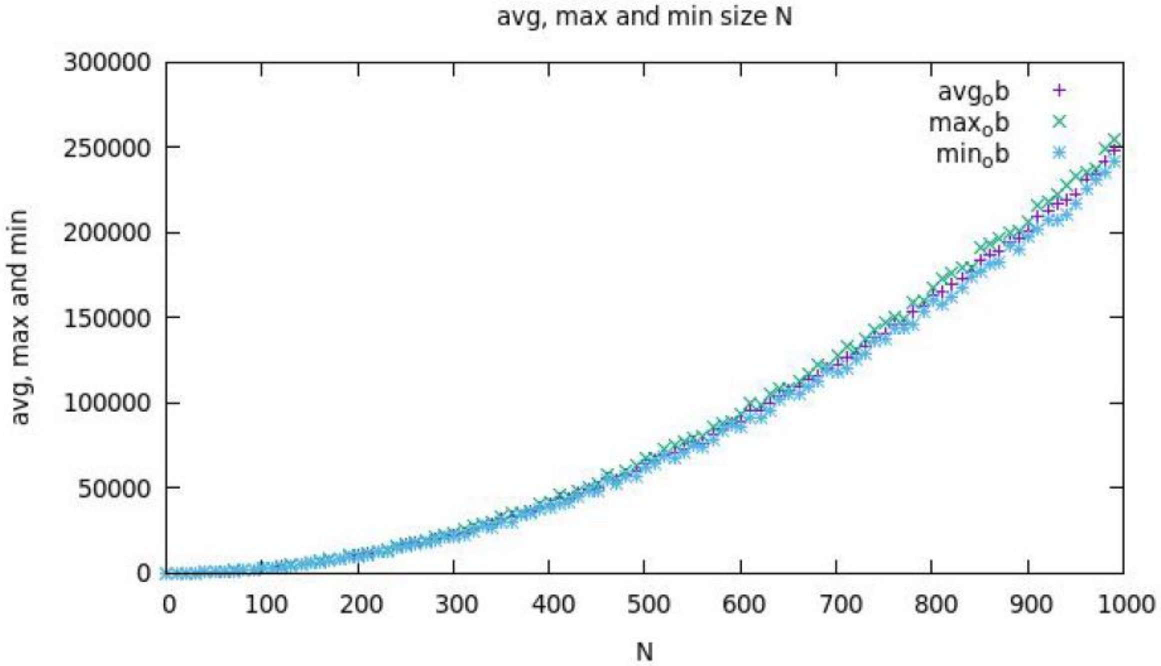
Same here

This plot is missing a title

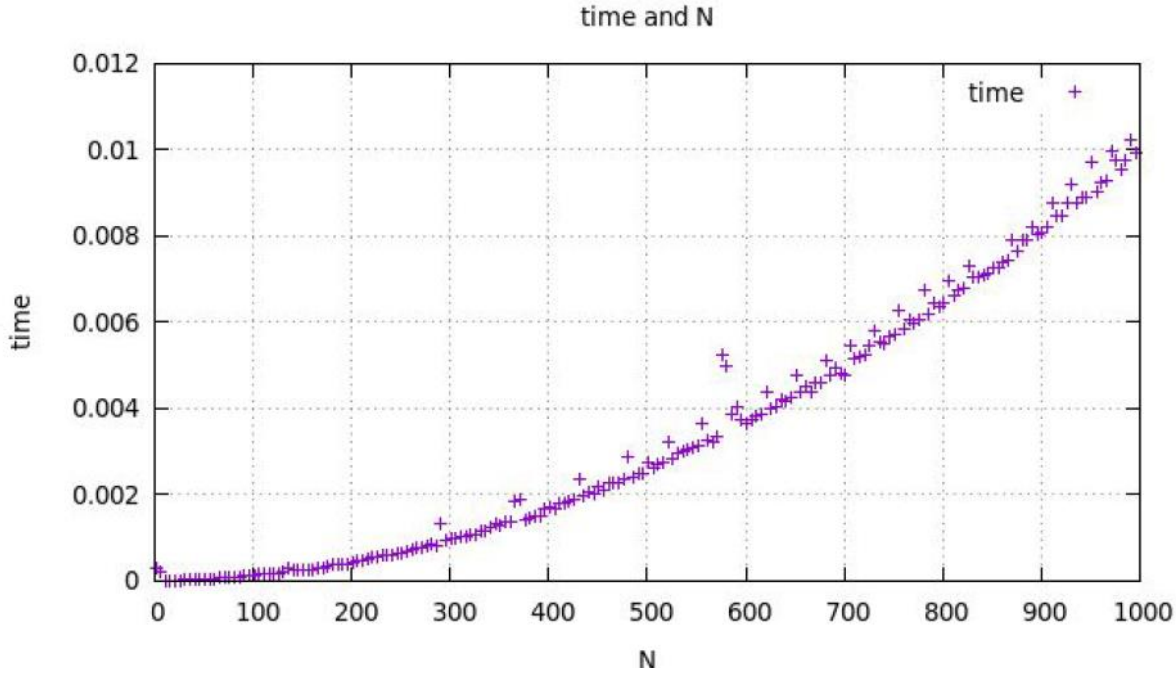
I'm missing the input parameters you used in exercise5.

In order to demostrate complexity is squared, you need to compare this plot with a square function (curve fitting). That will be mandatory in next lab.

5.6 Section 6



avg, max and min size N
avg, max and min
N
avg$_o$b
max$_o$b
min$_o$b



time and N
time
N

5. Answers to theoretical Questions.

-0.35

5.1 Question 1

We looked in the Internet information about how the rand function works and then we realized we had to change the range 0 - RANDMAX by using the formula:

```
random = (int) inf + rand() / ((RAND_MAX + 1.0) / (sup - inf + 1));
```

A little more explanation was expected

5.2 Question 2

This algorithm picks the second element of a table and after that it compares it with the one that has at his left.
If it is greater than the one that is in the second position it changes it. If not, the third element is picked, and it is compared with the second one. If the picked element is greater, it changes it and if not, it compares the chosen element with the following (the element that is on the left side of the one that it was compared to). It keeps going to the fourth and the rest of the table.
The element that its chosen is compared with the left element that it has. If the element of its left is greater it changes its position. If not, it goes to the left element of the one that has been compared with the chosen one.
The algorithm works well because it writes in each subtable and in each iteration the smallest element in the first position, after comparing it with the rest of the elements in the table, but just swapping it once.

5.3 Question 3

The outer loop of InsertSort does not act on the first element of the table because then it would not have an element on its left to compare with.

5.4 Question 4

It is a loop that executes in each operation:

No, it is only the comparison

```
while (j >= ip && table [j] > aux){
       table[j + i] = table[j];
       j--;
       cont++;
}
```

5.5 Question 5

W_{BS} (n) : InsertSort (σ)=(N^2 /2)+O(N)

B_{BS} (n) : InsertSort (σ) (N-1)

A little more explanation was expected

5.5 Question 6

As shown in the plots, the execution times for InsertSort and InsertSortInv are almost the same. This is because the order in which we want to sort the table does not affect the algorithm, because the mechanism is the same with the only difference that it looks for the biggest element instead of the smallest one.

6. Final Conclusions.

In general, this assignment has been interesting, because we first coded the algorithms to create tables of permutations and then we could see the results when doing the plots.

It was not something new for us because in theory class they have told as about the times and basic operations and before the practice we knew what the result was going to be, but it has been nice to see it and prove it by ourselves.