Analysis of Algorithms 2019/2020

Practice 1

José Manuel Freire and Rodrigo Juez, 1292.

Code	Plots	Memory	Total

1. Introduction.

At the beginning of the practise we discused about the way to generate equiprobable lists, because we could not use modulus operator, so we had to plan the operations we would use for the random numbers on paper. We were using to use the book Numerical recipes in C: the art of scientific computing, but we found out that the formula given there was not what we were looking for. After that we started with the first exercise.

2. Objectives

2.1 Section 1

Section 1 Objectives: in this first section we have to create a function that returns equiprobable numbers between a min and a max number given as arguments.

2.2 Section 2

Section 2 Objectives: in this second section we have to make a function that disorders a list of ordered numbers from 1 to N, being N the length of the list. The permutations have to be done swapping numbers randomly.

2.3 Section 3

Section 3 Objectives: In this section we have to generate “n_perms” lists of N numbers and test it with exercise3.c

2.4 Section 4

Section 4 Objectives: for the fourth section we have to create a function insertSort that sorts the elements of a list using the insert sort algorithm.

2.5 Section 5

Section 5 Objectives: We had to make a function called average_sorting_time which gives us a TIME_AA with the information we needed, one called generate_sorting_times, which does average_sorting_times for a set of lists, so we can compare for different Ns. Moreover, we have to make one last function to make a table in a file with all the obtained data, called save_time_table.

2.6 Section 6

Section 6 Objectives: For this section, we had to create an insertSortInv function that is an insert sort algorithm, but in the reverse order. So, the biggest numbers are the first.

3 Tools and Methodology

3.1 Section 1

We used Visual Studio Code with the plugin Live Share to work at the same time, and used Valgrind to check our memory usage. Sort and Uniq (added in the makefile) were used for checking the equiprobability and to generate an output usable for doing an histogram in excel. The testing was done generating 100000 random numbers from 1 to 10 in a loop just to make sure it worked correctly. We also generated random_numbers in the range from 100000 to 999999, so that when checking in the terminal is easier to spot a out of bounds mistake (if it would go under the -limInf we would see that one of the numbers generated had one less digit, which is easier to spot when you have 1000000000 numbers).

3.2 Section 2

We used Visual Studio Code's Live Share plugin to work at the same time as well as Valgrind to check any memory leak, or invalid read. We implemented the pseudocode specified in the pdf guide which was pretty straight forward.

Our testing methodology consisted of generating the biggest possible numbers until our computer didn't have enough memory (size 100000000 more or less) just to see if there was any overflow problem in Valgrind or memory leak.

3.3 Section 3

The tools were the same we used in the Section 2 albeit this time the testing didn't went as far in N as we also generated n_perms and the running time for 1000000 permutations with 1000 N would be too long. We didn't encounter any problems as the implementation was as easy as doing a loop generating permutations.

3.4 Section 4

Apart from our usual tools (Visual Studio's Live Share and Valgrind) we implemented the –g flag for debugging when our algorithm didn't work correctly. We used VS Code's debugger which is simple and fast to use. Testing was done by generating permutations of the same size as Section 2, but the time it took to finish was much longer. The development methodology consisted of understanding the algorithm and start programming until it behaved exactly as it was supposed to do, debugging was especially useful here as we could see the algorithm evolve through time to check if it worked correctly.

3.5 Section 5

This was by far the longest section, so we divided the work into several days, for this GitHub was useful. VS Code's debugger was also used for developing generate average_sorting_time, generate_sorting_times and save_time_table.

In Average sorting time we used the function clock_gettime which was the most precise one. As this function is not ANSI C we included a #define _POSIX_C_SOURCE 199309L which allowed us to use it.

In generate_sorting_times we didn't have any trouble as is just looping average_sorting_time. To calculate the size of the array of TIME_AAs we used another loop.

The function save_time_tables didn't cause any trouble. We made it automatically modify to the size of the input so that every number (N, clock_time, OBs...) would be in the same column.

3.6 Section 6

This section only consisted of modifying exercise5.c to send a pointer to InsertSortInv instead of InsertSort. This allowed us to create another table and to compare the times between the two algorithms.

4. Source code

4.1 Section 1

```
int random_num(int inf, int sup)
{
int x;
double d;
d=((double)rand())/((double)RAND_MAX + 1);
x=(int)(sup-inf+1)*d;
return (inf+x);
}
```

4.2 Section 2

```
int* generate_perm(int N)
{
  int i=0, swapindex=0, temp=0;
  int *permutations=NULL;
  permutations=(int*)malloc(sizeof(int)*N);
  if(!permutations) return NULL;

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

  for (i=0; i<N; i++){
      /*SWAP*/
    swapindex=random_num(i,N-1);
    temp = permutations[swapindex];
    permutations[swapindex] = permutations[i];
    permutations[i] = temp;
  }
  return permutations;
}
```

4.3 Section 3

```
int** generate_permutations(int n_perms, int N)
{
  int i;
  int** matrixPerm=NULL;

  matrixPerm=(int**)malloc(sizeof(int*)*n_perms);
  if(!matrixPerm) return NULL;
  for(i=0;i<n_perms;i++){
    matrixPerm[i]=NULL;
    matrixPerm[i]=generate_perm(N);
    if(!matrixPerm[i]) return NULL;
  }

  return matrixPerm;
}
```

4.4 Section 4

```
int InsertSort(int* table, int ip, int iu){
 int i, j, aux;
    int BOs=0;
    if(!table || ip < 0 || iu < 0 || ip > iu){
        printf("ERROR in InsertSort input");
        return ERR;
    }
   for(i = ip+1; i <= iu; i++){
      aux = table[i];

      for(j = i-1; j >= ip && table[j]>aux; j--){

        table[j+1]=table[j];

        BOs++;

      }

      table[j+1]=aux;

      BOs++;

    }

return BOs;

}
```

4.5 Section 5

```
void free_matrix(int** matrix, int sizea){
  int i=0;
  for(i=0; i<sizea; i++){
          free(matrix[i]);
        }
  free(matrix);
}

short average_sorting_time(pfunc_sort method,
                              int n_perms,
                              int N,
                              PTIME_AA ptime)
{
  int i, OBs=0;
  int **permutations=NULL;
  struct timespec start, end;
  long seconds, nanoseconds;
  double time=0, totalOBs=0;
  if(!method || !ptime || n_perms < 1 || N < 1) return ERR;
  permutations = generate_permutations(n_perms, N);
  if(!permutations) return ERR;
  ptime->N=N;
  ptime->n_elems=n_perms;
  ptime->min_ob=MAX_INT;
  for(i=0; i<n_perms; i++){

    if(clock_gettime(CLOCK_REALTIME, &start)==ERR){
      free_matrix(permutations, n_perms);
      permutations=NULL;
      return ERR;
    }
    OBs=method(permutations[i], 0, N-1);
    if(OBs==ERR) {
      free_matrix(permutations, n_perms);
      permutations=NULL;
      return ERR;
    }
     if(OBs>(ptime->max_ob)) ptime->max_ob=OBs;
    if(OBs<(ptime->min_ob)) ptime->min_ob=OBs;
    totalOBs+=OBs;
    OBs=0;

    if(clock_gettime(CLOCK_REALTIME, &end)==ERR){
      free_matrix(permutations, n_perms);
      permutations=NULL;
      return ERR;
    }

    seconds = end.tv_sec - start.tv_sec;
    nanoseconds = end.tv_nsec - start.tv_nsec;
    if (start.tv_nsec > end.tv_nsec) {
        --seconds;
        nanoseconds += 1000000000;
    }
    time+=seconds;
    time+=(float)nanoseconds/BILLION;

  }
    time=time/(double)n_perms;
    totalOBs=totalOBs/(double)n_perms;
    ptime->time=time;
    ptime->average_ob=totalOBs;

   free_matrix(permutations, n_perms);
  permutations=NULL;
  return OK;
}

short generate_sorting_times(pfunc_sort method, char* file,
      {
  int i=0, arraylenth=0, size_permutation=0;
  PTIME_AA times_array = NULL;

  if(!method || !file || num_min < 1 || num_max < num_min || incr < 0 ||
n_perms < 1) return ERR;
  if(incr != 0) for(i=num_min; i<=num_max; i+=incr) arraylenth++;
  else arraylenth = num_max - num_min + 1;
  size_permutation = num_min;
  times_array = (PTIME_AA)malloc(sizeof(TIME_AA)*arraylenth);
  if(!times_array) return ERR;
  /*INITIALIZE ARRAY*/
  for(i=0; i<arraylenth; i++){
    times_array[i].N = 0;
    times_array[i].n_elems = 0;
    times_array[i].time = 0;
    times_array[i].average_ob = 0;
    times_array[i].max_ob = 0;
    times_array[i].min_ob = MAX_INT;
  }
  for(i=0; i<arraylenth; i++){
    if(size_permutation > num_max){
      free(times_array);
      times_array = NULL;
      return ERR;
    }
    if(average_sorting_time(method, n_perms, size_permutation,
&(times_array[i])) == ERR) {
      free(times_array);
      return ERR;
    };
    size_permutation += incr;

  }
  save_time_table(file, times_array, arraylenth);


  free(times_array);
  return OK;
}
/*N average_clock_time, average_ob, max_ob, min_ob*/
short save_time_table(char* file, PTIME_AA ptime, int n_times)
{
  FILE *save = NULL;
  int i=0, space_n, space_obs, n_calculator=0, ob_calculator=0, total=0;
  char buffer[MAX_CHAR];

  if(!file || !ptime || n_times < 1) return ERR;
  save = fopen(file,"w");
  if(!save) return ERR;

  for(i=0; i<n_times; i++){
    if(n_calculator < ptime[i].N) n_calculator = ptime[i].N;
    if(ob_calculator < ptime[i].max_ob) ob_calculator = ptime[i].max_ob;
  }
  /*calculate space for N*/
  for(space_n=0; n_calculator != 0; space_n++) n_calculator =
n_calculator/10;
  space_n++;
  /*calculate space for OBs*/
  for(space_obs=0; ob_calculator != 0; space_obs++) ob_calculator =
ob_calculator/10;
  space_obs++;

  if(space_obs<6) space_obs=6;
  fprintf(save, "\n");
  /*HEATHER*/
    sprintf(buffer, "|N");
    total+=2;
    total = strlen(buffer);
    while(strlen(buffer)<space_n){
      strcat(buffer, " ");
      total++;
    }
    strcat(buffer, "|");
    total++;
    fputs(buffer, save);
    sprintf(buffer, "clock_time");
    strcat(buffer, "|");
    total+=11;
    fputs(buffer, save);

    sprintf(buffer, "AVERAGE_OB");
    total += 10;
    while(strlen(buffer)<(space_obs+6)){
      strcat(buffer, " ");
      total++;
    }
    strcat(buffer, "|");
    total++;
    fputs(buffer, save);

    sprintf(buffer, "MAX_OB");
    total += 6;
    while(strlen(buffer)<(space_obs)){
      strcat(buffer, " ");
      total++;
    }
    strcat(buffer, "|");
    total++;
    fputs(buffer, save);

    sprintf(buffer, "MIN_OB");
    total += 6;
    while(strlen(buffer)<(space_obs)){
      strcat(buffer, " ");
      total++;
    }
    strcat(buffer, "|\n");
    total++;
    fputs(buffer, save);

  /*DIVISOR*/
    for(i=0; i<total; i++){
      fprintf(save, "_");
    }
    fprintf(save, "\n");

  for(i=0; i<n_times; i++){
    /*N:*/
    sprintf(buffer, "|%d", ptime[i].N);
    while(strlen(buffer)<space_n){
      strcat(buffer, " ");
    }
    strcat(buffer, "|");
    fputs(buffer, save);

    /*AVERAGE_CLOCK_TIME*/
    sprintf(buffer, "%f  ", ptime[i].time);
    strcat(buffer, "|");
    fputs(buffer, save);
```

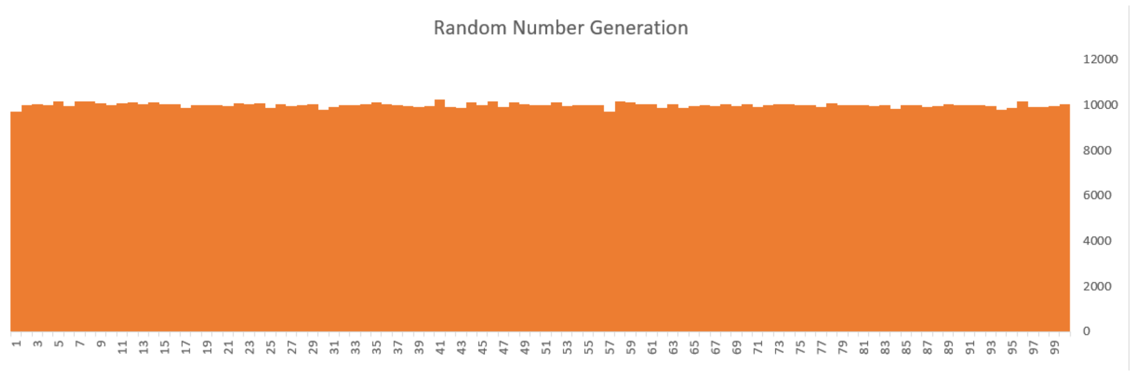
```
    /*AVERAGE_OB*/
    sprintf(buffer, "%f", ptime[i].average_ob);
    while(strlen(buffer)<(space_obs+6)){
      strcat(buffer, " ");
    }
    strcat(buffer, "|");
    fputs(buffer, save);

    /*MAX_OB*/
    sprintf(buffer, "%d", ptime[i].max_ob);
    while(strlen(buffer)<space_obs){
      strcat(buffer, " ");
    }
    strcat(buffer, "|");
    fputs(buffer, save);

    /*MIN_OB*/
    sprintf(buffer, "%d", ptime[i].min_ob);
    while(strlen(buffer)<space_obs){
      strcat(buffer, " ");
    }
    strcat(buffer, "|\n");
    fputs(buffer, save);
  }
  /*DIVISOR*/
    for(i=0; i<total; i++){
      fprintf(save, "¯");
    }
    fprintf(save, "\n");
   fclose(save);
  return OK;
}
```

4.6 Section 6

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

    int i, j, aux;
    int BOs=0;
    if(!table || ip < 0 || iu < 0 || ip > iu){
        printf("ERROR in InsertSort input");
        return ERR;
    }

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

    return BOs;

}
```

5. Results, Plots

5.1 Section 1



This is the result of generating 1000 random numbers from 1 to 10, and as we can see the result obtained is very closed to the equiprobability and when generating more than 1000 numbers we obtained an even more plain histogram, but it took more time to compute.

5.2 Section 2

We tested generating 10000 permutations of size 1000 and they were generated correctly, and without valgrind errors, we also tested generating smaller arrays to manually check the key points such as the start and the end would be generated correctly.

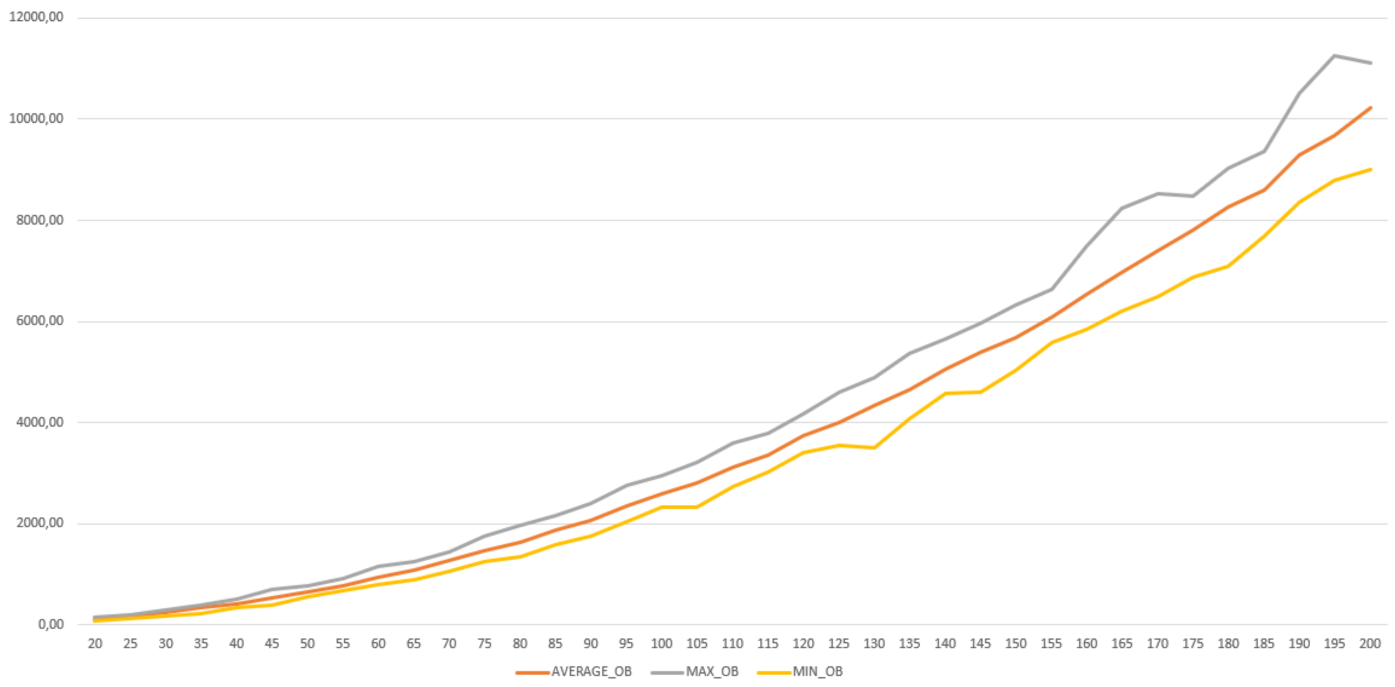
5.3 Section 3

We ran the exact same tests as for section 2, and worked as well. Essentially they do the same but this exercise performs the permutations in the function genera_permutations instead of a loop in the main.

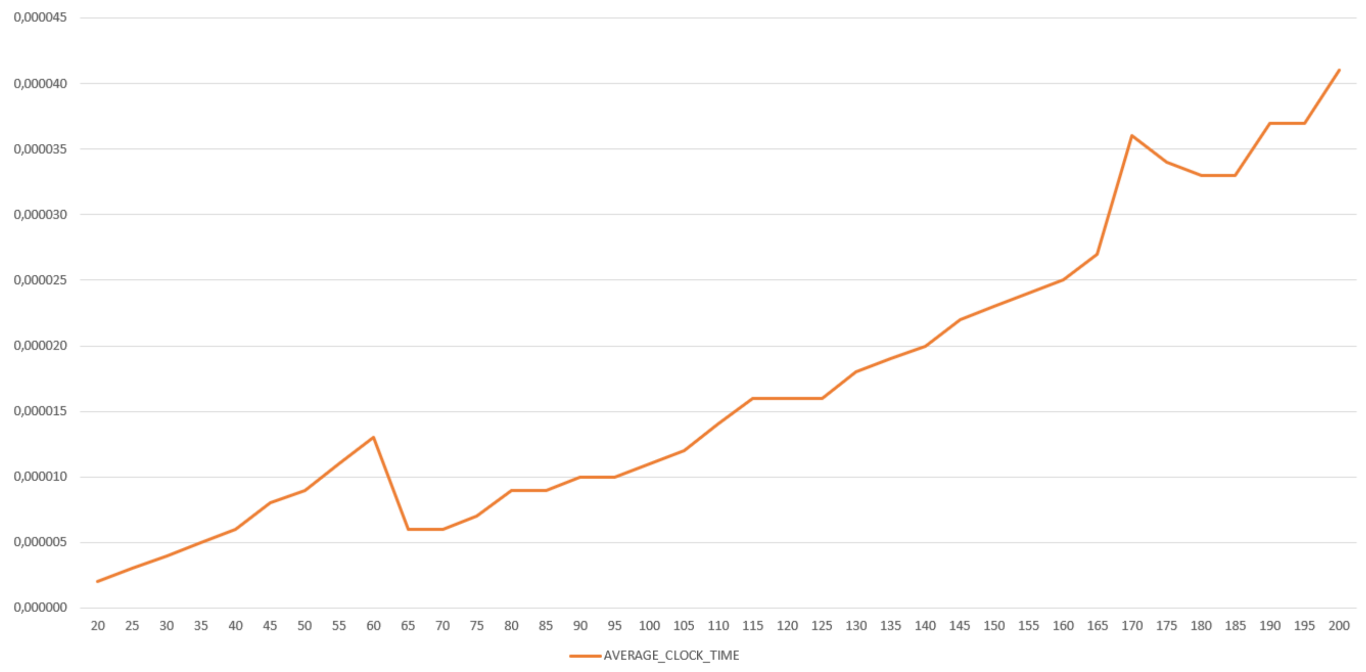
5.4 Section 4

We ran tests of sizes in the range of 10 to 100000 to check if there were any memory errors but, there weren't and they were successfully sorted.

5.5 Section 5

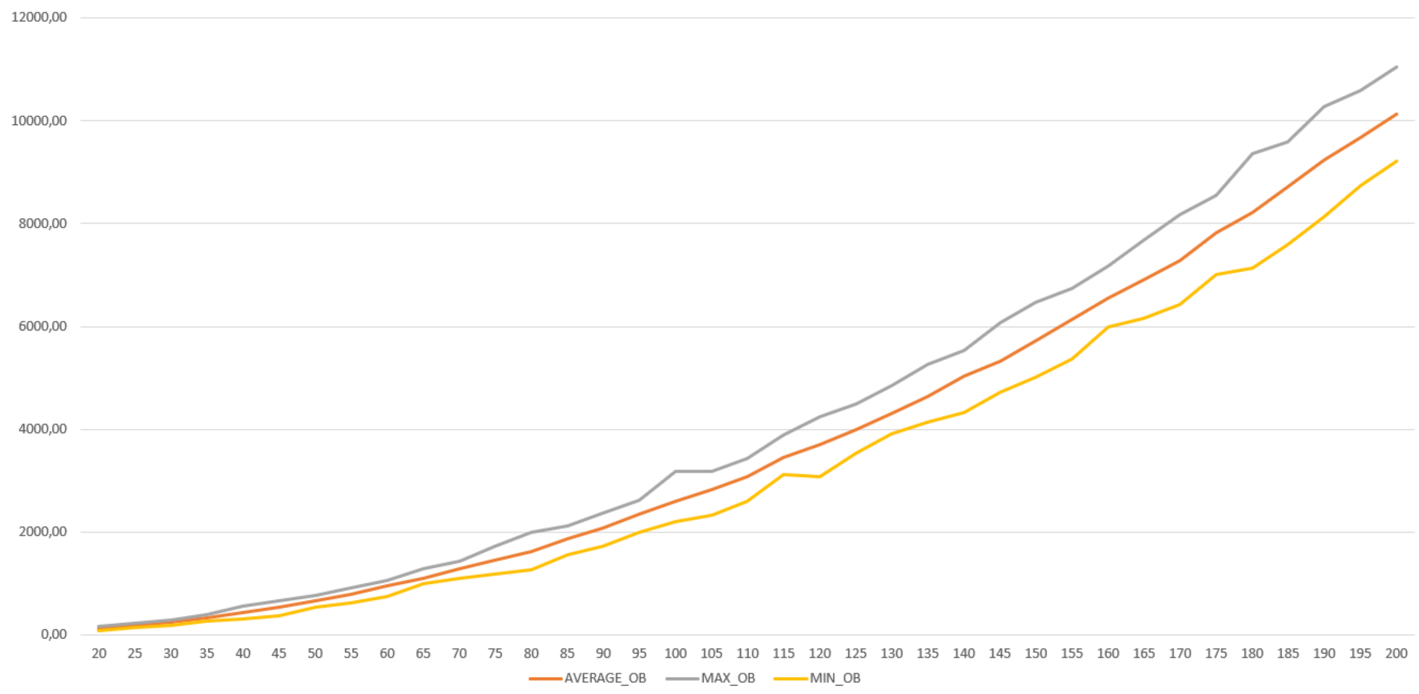


It can clearly be seen that in InsertSort algorithm the summation growth is of O(N^2) and that as we grow, the minimum and maximum separate more, as there is more room for being random.

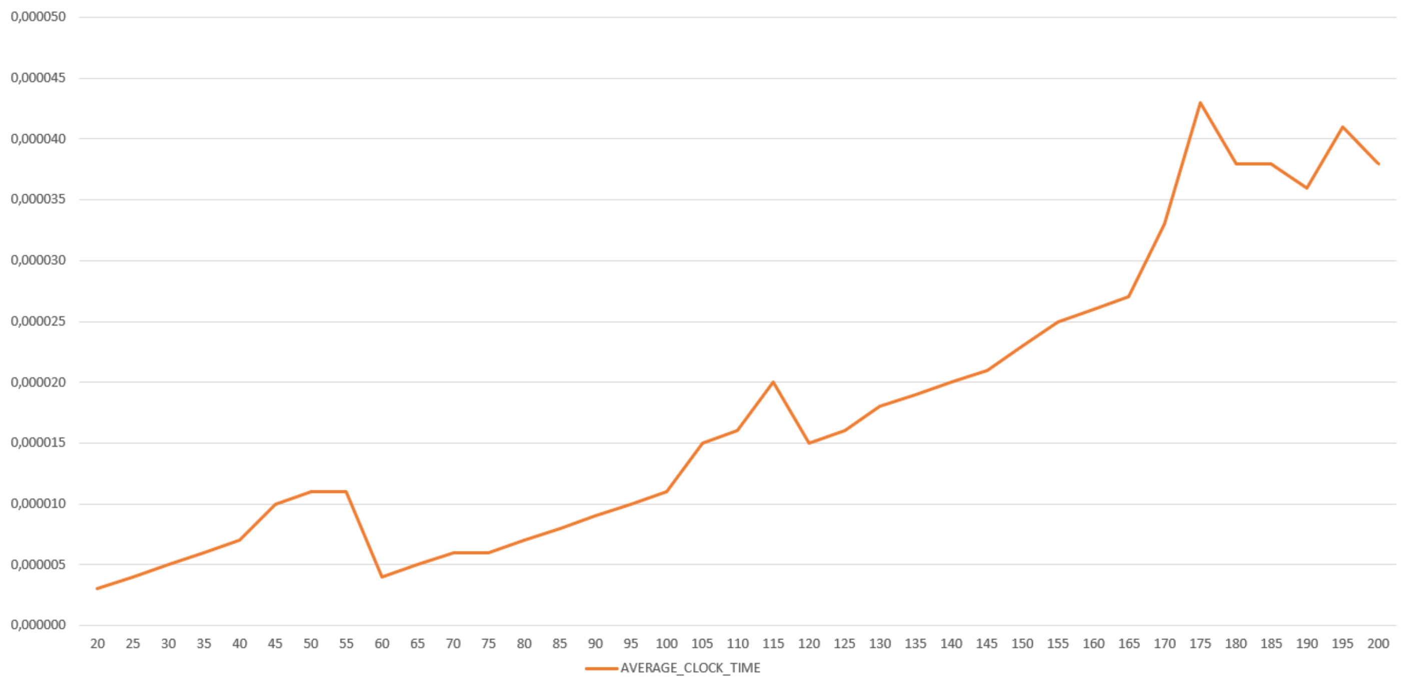


Here it can be seen also that grows in O(N^2) but as it depends more on the load of the CPU when running, and we used a laptop for testing, it’s heavily influenced by the boost of the CPU also.

5.6 Section 6



The same conclusions can be obtained from the InsertSortInv as it’s essentially the same algorithm. There is some room for error between InsertSort and InsertSortInv as when running two different permutations in each loop were used but this was contra rested by using a high number of permutations so that the average would be precise.



We can see a similar trend in the clock_times but as we specified earlier this varies a lot as it’s dependant on the CPU load, temperature, boost…

5. Answers to theoretical Questions.

5.1 Question 1

We obtained a formula from the book Numerical recipes in C: the art of scientific computing, but it was designed to work with floating point limits, as our limits where integer, we modified slightly the function adding castings. When we ran the formula, we obtained out of boundaries random numbers so then we modified the formula to work in the boundaries. Another way to obtain a random number would be with the standard function and then applying the module to make it cyclical, this is worse as it can be predicted more easily.

5.2 Question 2

InsertSort is a local algorithm which only makes one key comparison at a time. It works by considering the first element of the unsorted array already sorted and then each iteration it removes one element of the array (the next-one) and moves the previous one until it finds it's place where it would be sorted.

5.3 Question 3

Because it assumes that is already sorted, as in the rest of the iterations it will be moved forward in case there is a smaller number.

5.4 Question 4

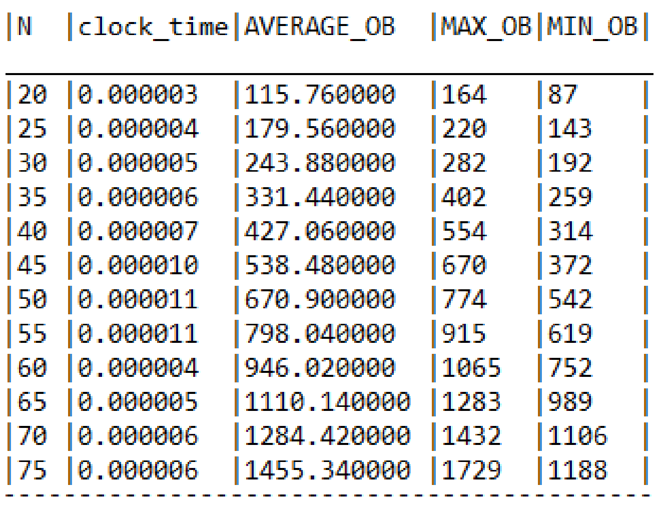
The basic operation is table[j]>aux because is in the most inner loop.

5.5 Question 5

1. The average case for InsertSort is $N^2/4 + O(N)$
2. The worst case is $\leq N^2/2 + O(N)$
3. The best case is N-1

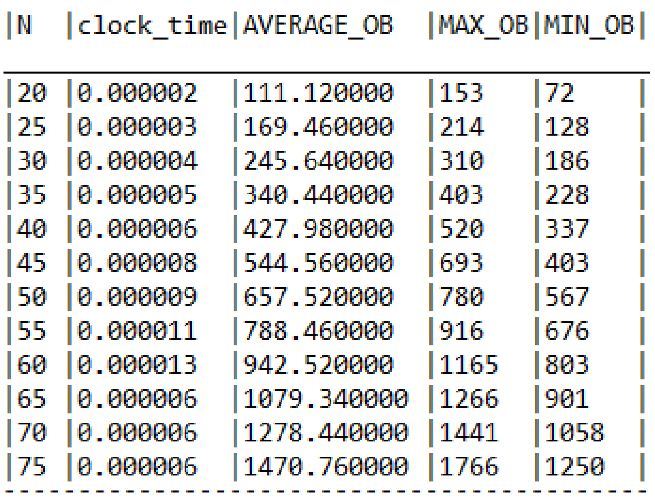
5.5 Question 6

We generated 2 tables with 50 permutations in each run one with the method InsertSort and the other with InsertSortInv, the results were very similar as we can see. We think that this is due to the different permutations in each run.



```
|N  |clock_time|AVERAGE_OB  |MAX_OB|MIN_OB|

|20 |0.000003  |115.760000  |164   |87    |
|25 |0.000004  |179.560000  |220   |143   |
|30 |0.000005  |243.880000  |282   |192   |
|35 |0.000006  |331.440000  |402   |259   |
|40 |0.000007  |427.060000  |554   |314   |
|45 |0.000010  |538.480000  |670   |372   |
|50 |0.000011  |670.900000  |774   |542   |
|55 |0.000011  |798.040000  |915   |619   |
|60 |0.000004  |946.020000  |1065  |752   |
|65 |0.000005  |1110.140000 |1283  |989   |
|70 |0.000006  |1284.420000 |1432  |1106  |
|75 |0.000006  |1455.340000 |1729  |1188  |
----------------------------------------
```

InsertSortInv



```
|N  |clock_time|AVERAGE_OB  |MAX_OB|MIN_OB|

|20 |0.000002  |111.120000  |153   |72    |
|25 |0.000003  |169.460000  |214   |128   |
|30 |0.000004  |245.640000  |310   |186   |
|35 |0.000005  |340.440000  |403   |228   |
|40 |0.000006  |427.980000  |520   |337   |
|45 |0.000008  |544.560000  |693   |403   |
|50 |0.000009  |657.520000  |780   |567   |
|55 |0.000011  |788.460000  |916   |676   |
|60 |0.000013  |942.520000  |1165  |803   |
|65 |0.000006  |1079.340000 |1266  |901   |
|70 |0.000006  |1278.440000 |1441  |1058  |
|75 |0.000006  |1470.760000 |1766  |1250  |
----------------------------------------
```

InsertSort

6. Final Conclusions.

We found out that insert sort is good for permutations with only a few numbers, but when the number gets bigger, the runtime grows much faster because its growth is $O(N^2)$. It’s as fast as a local algorithm can be but still slow for big inputs. Sometimes it even took more than 10 minutes to run in our computers with huge numbers.