Analysis of Algorithms 2020/2021

Practice 1

Kevin de la Coba and Marcos Bernuy, group 1292, pair 2.

Code	Plots	Memory	Total

1. Introduction.

In this assignment we measured the behavior of the insert sort algorithm, for that we used C creating a data structure to store all the info related with the performance of the algorithm with different permutations that can have different size.

2. Objectives

2.1 Section 1

We built a function which can generate random numbers given the smallest and the biggest to generate.

2.2 Section 2

We built a function which can generate a permutation using the previous function that generates a random number.

2.3 Section 3

We built a function which basically generates an array of permutations.

2.4 Section 4

We built the insert sort function based on the pseudocode of the theory PowerPoint; we also return basic operations (in this case key comparisons) that the algorithm does to solve a given permutation.

2.5 Section 5

We built a module which measures the time that a given algorithm spends sorting an array and it measures the basic operations done.

2.6 Section 6

We built a modification of the insert sort function, in this case the algorithm sorts a permutation in reverse order, this means that the permutations ends having the biggest value at the beginning and the smallest one at the end.

3 Tools and Methodology

3.1 Section 1

We used Visual studio code and we tested the functions in ubuntu. In order to solve this section we built a function that given two limits (up and down limits), it generates random numbers between those 2 limits.

3.2 Section 2

Given a size N we create an array of that size and we fill it with numbers in ascending order (1, 2, …, N), then we use the function we created in the previous section and we swap every element with a random one (for example, i = 0, swapped with j = random_num), we swap every number with a random index of the array.

3.3 Section 3

For this section we just create an array of arrays that contains "n_perms" permutations.

3.4 Section 4

We took the algorithm insert_sort from the theory and we built the function that sorts using that algorithm.

3.5 Section 5

We built the functions that stores data in the TIME_AA structure and then it stores this data in a file.

3.6 Section 6

We just changed a "<" to ">", so now the function sorts the array in reverse order.

4. Source code

4.1 Section 1

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

  int random = 0;

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

  random = rand();
  if(random < 0) return ERR;

  /* Taking the random number between 0 and rand_max */
  random = inf + (int) (((sup - inf + 1.0) * random) / (RAND_MAX + 1.0));
  if (random < inf || random > sup) return ERR;

  return random;
}
```

4.2 Section 2

```c
int* generate_perm(int N) {

  int i, random;
  int *perm = NULL;

  if(N <= 0)
    return NULL;

  /* Reserving memory for the permutation */
  perm = (int*)malloc(N * sizeof(int));
  if (perm == NULL)
    return NULL;

  /* Filling the documentation */
  for(i = 0; i < N; i++){
    perm[i] = i+1;
  }

  /* Swaping numbers in the permutation */
  for(i = 0; i < N; i++) {
    random = random_num(i, N-1);
    if (random == ERR) {
      free(perm);
      return NULL;
    }
    if (swap(&perm[i], &perm[random]) != OK){
      free(perm);
      return NULL;
    }
  }

  return perm;
}
```

4.3 Section 3

```c
int** generate_permutations(int n_perms, int N) {
  int i = 0, n = 0, **perms = NULL;

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

  /* Reserving memory for the permutations */
  perms = (int**)malloc(n_perms*sizeof(int*));
  if (perms == NULL) return NULL;

  /* Loop for creating the permutations */
  for (i = 0; i < n_perms; i++) {

    perms[i] = generate_perm(N);
    if (perms[i] == NULL) {
      for(n = 0; n < i; n++) free(perms[n]);
      free(perms);
      return NULL;
    }
  }

  return perms;
}
```

4.4 Section 4

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

  int i = 0, j = 0, A = 0, counter = 0;

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

  for (i = ip+1; i <= iu; i++) {
    A = table[i];
    j = i-1;
    while(j >= ip) {
      counter++;
      if (table[j] > A) {
        table[j+1] = table[j];
        j--;
      }
      else break;
    }
    table[j+1] = A;
  }
  return counter;
}
```

4.5 Section 5

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

  clock_t start = 0, end = 0;
  double total = 0;
  int **perms = NULL, i = 0, n = 0;
  long BOs = 0, total_BOs = 0;

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

  /* Creating all the permutations */
  perms = generate_permutations(n_perms, N);
  if (perms == NULL) return ERR;

  start = clock();
  if (start == (clock_t) -1) {
    for (i = 0; i < n_perms; n++) free(perms[i]);
    free(perms);
    return ERR;
  }

  /* Loop to sort all the permutations */
  for (i = 0; i < n_perms; i++) {

    /* Using the function to sort the array */
    BOs = method(perms[i], 0, N-1);
    if (BOs == ERR) {
      for (n = 0; n < n_perms; n++) free(perms[n]);
      free(perms);
      return ERR;
    }

    /* Checking the maximum and minimum values of the structure */
    if (ptime->min_ob == 0 || ptime->min_ob > BOs) ptime->min_ob = BOs;
    if (ptime->max_ob == 0 || ptime->max_ob < BOs) ptime->max_ob = BOs;

    total_BOs += BOs;
  }

  end = clock();
  if (end == (clock_t) -1) {
    for (i = 0; i < n_perms; n++) free(perms[i]);
    free(perms);
    return ERR;
  }

  /* Assingning the values to the structure */
  total = ((double) (end - start)) / CLOCKS_PER_SEC;
  ptime->time = total / n_perms;
  ptime->N = N;
  ptime->n_elems = n_perms;
  ptime->average_ob = ((double) total_BOs)/n_perms;
  for(i = 0; i < n_perms; i++) free(perms[i]);
  free(perms);

  return OK;
}
```

```c
short generate_sorting_times(pfunc_sort method, char* file,
                                int num_min, int num_max,
                                int incr, int n_perms) {
  int i = 0, counter = 0, n = 0;
  PTIME_AA ptime = NULL;

  /* Checking arguments */
  if (method == NULL || file == NULL || num_min < 0 || num_max < 0 || incr < 1 ||
n_perms < 1)
    return ERR;

  /* Allocating memory for the array of ptimes */
  counter = ((num_max - num_min) / incr) + 1;
  ptime = (PTIME_AA)malloc(counter * sizeof(TIME_AA));
  if (ptime == NULL)
    return ERR;

  /* Initializing ptime */
  for(i = 0; i < counter; i++) {
    ptime[i].average_ob = 0;
    ptime[i].max_ob = 0;
    ptime[i].min_ob = 0;
    ptime[i].N = 0;
    ptime[i].n_elems = 0;
    ptime[i].time = 0;
  }

  /* Calling the function to get the ptimes */
  for(i=num_min; i <= num_max; i += incr, n++){
    if(average_sorting_time(method, n_perms, i, &ptime[n]) == ERR){
      free(ptime);
      return ERR;
    }
  }

  /* Calling save_time_table so the information is displayed in the file */
  if (save_time_table(file, ptime, counter) == ERR){
    free(ptime);
    return ERR;
  }

  free(ptime);
  return OK;
}
```

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

  int i = 0;
  FILE *fp = NULL;

  /* Checking arguments */
  if (file == NULL || ptime == NULL || n_times < 0) return ERR;

  /* Opening the file */
  fp = fopen(file, "w");
  if (fp == NULL) return ERR;

  /* Printing all the ptimes */
  for (i = 0; i < n_times; i++){

    fprintf(fp, "Number of elements: %d\t"
            "Size of each element: %d\t"
            "Average clock time: %e\t"
            "Average OB executed: %f\t"
            "Minimun OB executed: %d\t"
            "Maximun OB executed: %d\t\n", ptime[i].n_elems, ptime[i].N,
ptime[i].time, ptime[i].average_ob, ptime[i].min_ob, ptime[i].max_ob);
  }

  /* Closing the file */
  fclose(fp);

  return OK;
}
```

4.6 Section 6

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

  int i = 0, j = 0, A = 0, counter = 0;

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

  for (i = ip + 1; i <= iu; i++){
    A = table[i];
    j = i - 1;
    while (j >= ip){
      if(table[j] < A){
        counter++;
        table[j+1] = table[j];
        j--;
      } else break;
    }
    counter++;
    table[j+1] = A;
  }
  return counter;
}
```

5. Results, Plots

5.1 Section 1

The console output is this one:

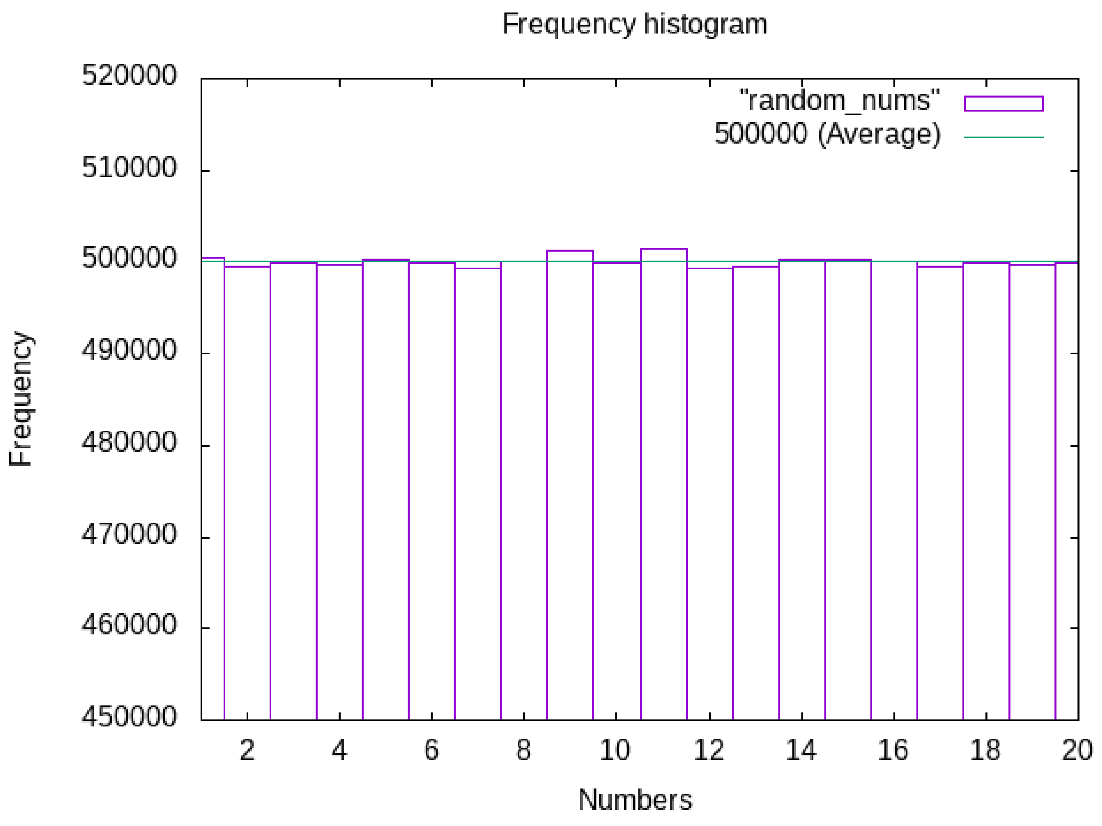
```
./exercise1 -limInf 1 -limSup 20 -numN 20

Running exercise1

Practice no 1, Section 1

Done by: Kevin de la Coba and Marcos Bernuy

Grupo: 1292 - T02

17

9

8

20

16

14

11

19

17

6

19

7

6

19

12

2
```

14

18

4

1



This is the plot after we executed the exercise1 with the following arguments:

```
      -limInf 1 -limSup 20 -numN 10000000
```

As we can see in the plot every number is generated the same amount of times. So we can conclude that all the numbers are going to be generated:

“times generated/numbers”

That is the same as the average, this confirms the *law of big numbers*.

5.2 Section 2

```
./exercise2 -size 15 -numP 10

Running exercise2

Practice number 1, section 2

Done by: Kevin de la Coba and Marcos Bernuy

Grupo: 1292 - T02

8 3 1 13 11 10 5 9 4 14 12 6 2 7 15

1 2 4 6 14 7 13 12 9 11 8 15 10 5 3

14 12 2 7 11 9 13 8 3 6 15 5 4 1 10

6 4 2 9 8 15 11 5 12 13 3 1 14 10 7

3 15 13 14 8 10 9 5 12 4 1 11 2 7 6

13 14 11 15 8 4 2 6 9 12 1 3 10 7 5

6 4 7 1 5 12 2 10 13 8 11 3 9 15 14

12 1 11 8 2 15 10 5 6 4 9 13 14 7 3

1 11 10 8 14 12 7 2 15 5 13 6 9 4 3

9 10 13 6 15 2 8 4 1 7 14 12 11 3 5
```

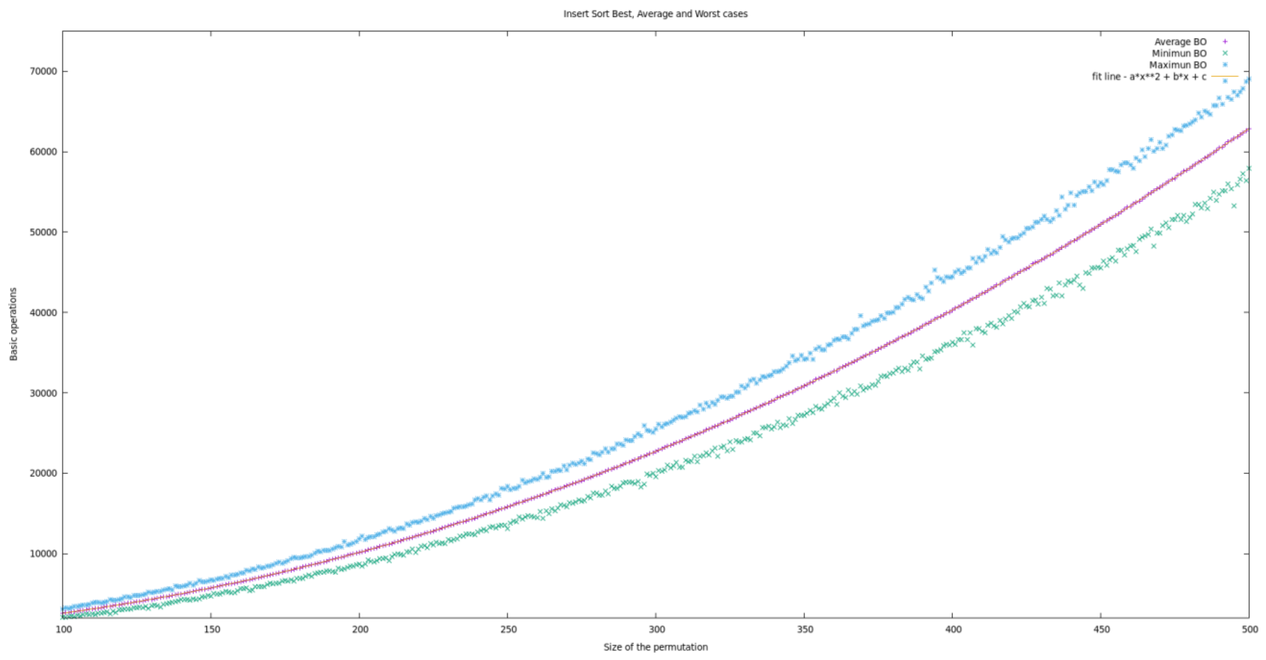
5.3 Section 3

```
./exercise3 -size 15 -numP 10

Running exercise3

Practice number 1, section 3

Done by: Kevin de la Coba and Marcos Bernuy

Grupo: 1292 - T02

4 9 3 8 11 6 2 1 14 15 5 12 10 13 7

13 15 3 9 1 5 4 7 10 6 14 11 8 12 2

14 6 4 1 12 5 15 8 3 9 7 11 2 13 10

5 9 10 7 15 2 8 3 12 6 1 4 11 14 13

9 2 1 11 14 7 3 5 10 6 8 12 15 4 13

9 12 6 5 4 8 11 1 2 15 13 14 7 10 3

4 3 11 12 1 2 15 8 6 10 9 7 13 14 5

13 11 4 5 14 8 10 7 2 15 9 6 3 1 12

2 9 10 13 7 8 1 5 11 14 12 4 3 6 15

1 4 5 6 8 2 10 13 11 14 7 9 15 12 3
```

5.4 Section 4

```
./exercise4 -size 5

Running exercise4

Practice number 1, section 4

Done by: Kevin de la Coba and Marcos Bernuy

Grupo: 1292 - T02

5	4	3	2	1

BO: 10

1	2	3	4	5
```

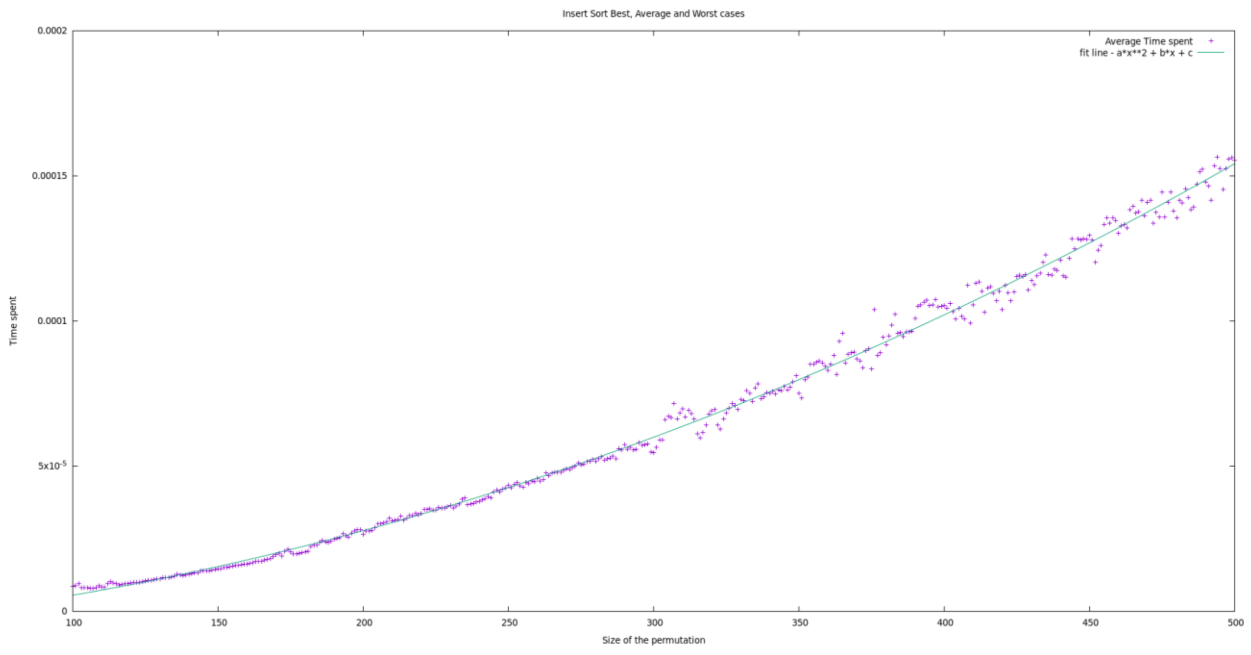
5.5 Section 5



Here we can see the plot generated after executing the exercise5 with the following arguments:

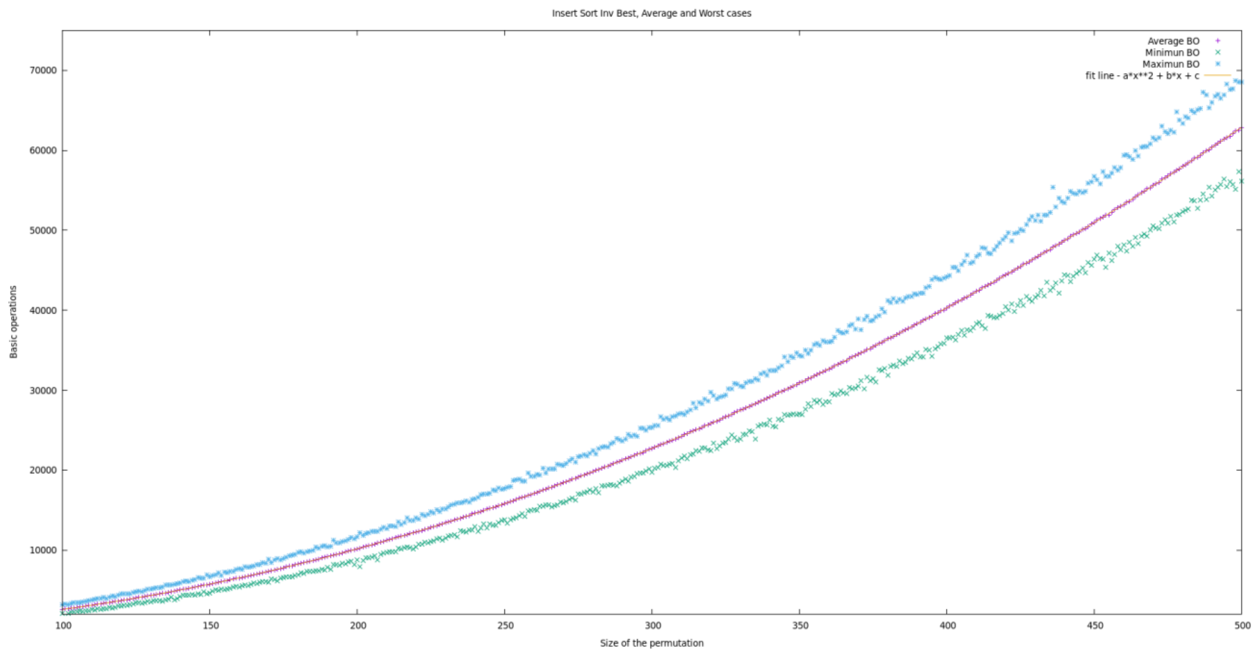
```
    @./exercise5 -num_min 100 -num_max 500 -incr 1 -numP 1000 -
outputFile exercise5.log
```

We can see that the behavior of the growth of the basic operations is not linear, is quadratic. We can also see that the best case is always below the average case and that the worst case is always above the average case. We can also see that the average case fits a quadratic behavior so well that we can barely see the fit line under the data of the average basic operations.



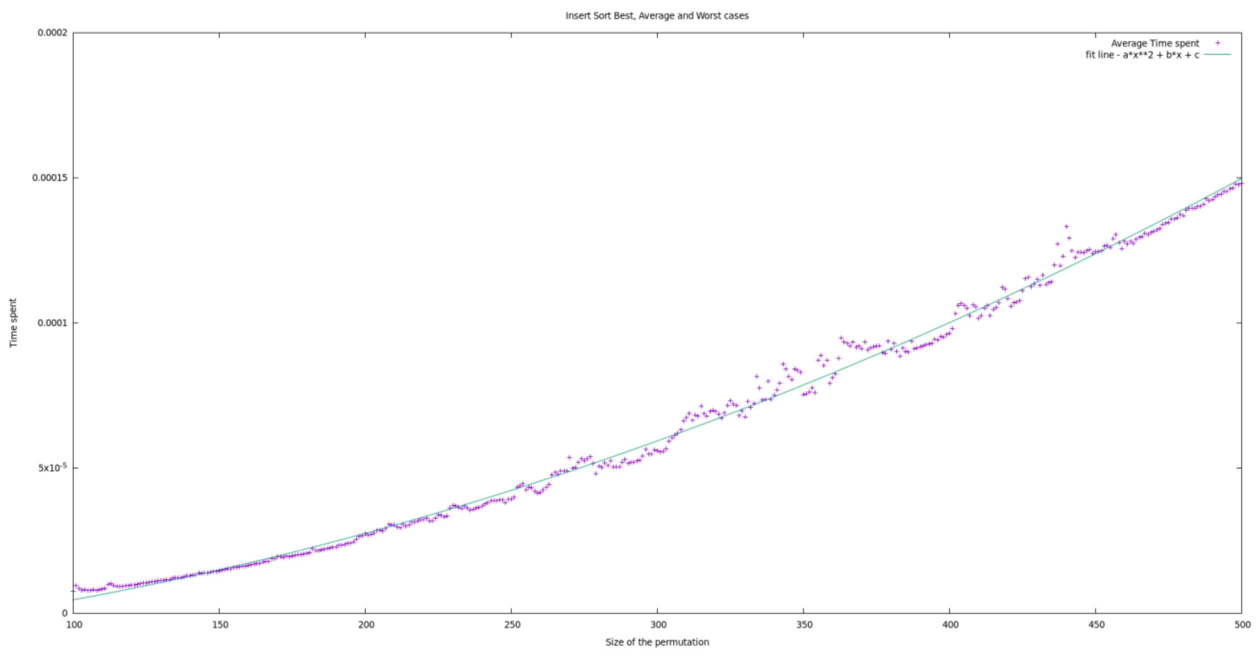
In this plot we can see that the way that the time grows is also quadratic but, in this plot the data is more “separated” that is because the CPU may do another tasks while we ran our program. The important thing is that the time growths also in a quadratic way.

5.6 Section 6



In order to get this plot we previously created the file “exercise5_inv.log” following the process of executing the exercise5.c with its corresponding parameters.

In this plot we can see that it behaves exactly like the algorithm non-inverted. That means it keeps the same quadratic behavior, keeping the best case or the one with the least basic operations always below the average and worst case. It also has the fit line situated exactly where the average case is.



In this plot we can see that in the lowest and highest size of permutations the time spent is more constant whereas in the medium sizes the values are more spread.

5. Answers to theoretical Questions.

Here you answer to the theoretical questions in the practice.

5.1 Question 1

In our code we used the function rand() which is a pseudo-random function and therefore it isn't enough to create a random number. To solve this problem, we decided to construct a combined generator. We combined different methods which are completely different between them. First method is the addition of the inferior number and (int), then there's the multiplication of rand() with the subtraction of the superior number with the inferior number plus one. And finally the multiplication of those two divided by RAND_MAX + 1. As you can see each term of the algorithm is clearly different from the rest. We had a little inspiration by the book "Numerical recipes in C: the art of scientific computing", but we mostly got inspired by last year's implementation of the random function.

5.2 Question 2

The algorithm works well because it iterates in every element of the array, and it compares each element with the previous one, if the number compared is smaller than the previous one this element is swapped with the previous one and then we compare again the element with his new previous element until we find a number that is not smaller than our element. If we consider that we are doing this for each element in the array, we are going to see how the algorithm leaves the elements of the array that it has checked sorted, so when it reaches the end, the array is sorted.

5.3 Question 3

The outer loop of the algorithm doesn’t act on the first element because the number is used as key number and it is implied that the number is ordered. The number to be compared will always have at its left of the array the numbers ordered and it will go through them until fitting correctly in its place. This would be an example of the insert sort algorithm:

$9\ 6\ 3\ 2 \rightarrow 6\ 9\ 3\ 2 \rightarrow 6\ 3\ 9\ 2 \rightarrow 3\ 6\ 9\ 2 \rightarrow 3\ 6\ 2\ 9 \rightarrow 3\ 2\ 6\ 9 \rightarrow 2\ 3\ 6\ 9$

In the example we can see that the number most situated at the left would be the last possible key number to be used.

5.4 Question 4

The basic operation of the algorithm would be the comparison table[j] > A, so the comparison of the numbers ordered in the table, and the number used as key or number which is being ordered.

5.5 Question 5

In order to get the best and worst case, we must know how the work on the innermost loop is which depends on the input.

$W_{IS}(N)$: $N^2/2 + O(N)$

$A_{IS}(N)$: $O(N^2)$

$B_{IS}(N)$: $N-1$

5.5 Question 6

Looking at the graphs which define both algorithms, and also checking the values, we can clearly see that both algorithms are quadratic and that they develop the same way. This is understandable since the algorithm follows the same path no matter if we are looking for an array ordered from biggest number to smallest or all the way around. When implementing the algorithm the only change made is the sign of the key comparison. Instead of being ‘<’ it becomes ‘>’. So the path, the time needed, and the worst, best and average cases don’t change at all from one algorithm to the other.

6. Final Conclusions.

To conclude this report, we can say that we are applying the scientific method and we are measuring the performance of insert sort. Checking the results, we can confirm what we learnt in theory, the insert sort time and basic operations that the algorithm needs to sort a permutation grows with the size of the array, and it does in a quadratic way $O(n^2)$.