Analysis of Algorithms 2019/2020

Practice 1

Paula Samper López

Laura de Paz Carbajo

18 / 10 / 2019

Group 125

Code	Plots	Memory	Total

1. Introduction.

In the first practice of this course we have put into practice the knowledge we have acquired in the theoretical lessons during the first lectures of this course. For instance, we have implemented a sorting algorithm, worked with it and later showed its worst, average and best case as previously studied in class.

2. Objectives

2.1 Section 1

This section´s goal was to implement the function *int random num (int inf, int sup)* to generate a random number between a maximum and a minimum, without using module "%". We had to add this routine to permutations.c, check the results by running exercise1.c and creating a histogram.

2.2 Section 2

In this section we were provided with a pseudocode that generated random permutations. The first part consisted on a loop that created an array or sorted elements from 1 to N, and the second part generated N random numbers that were later used as indexes to swaps random elements within the array. Thus, we get a random permutation of N elements with numbers from 1 to N.

Our task is to implement in C code the function *int * generate perm (int N)* that works as the pseudocode provided, and then check its correctness executing exercise2.c.

2.3 Section 3

Our task was to create the function *int ** generate permutations (int n perms, int N)* to create n random permutations of N using the function *int * generate perm (int N)* from the previous section. We had to end up with an array, and each array element was a pointer to one of these permutations. We checked the correctness of the function by executing exercise3.c.

2.4 Section 4

This section´s goal was to implement the routine *int InsertSort (int * table, int ip, int iu)* to replicate the InsertSort method (*table* is the table to sort, *ip* is the first element, and *iu* is the last one). The function had to return the number of basic operations. We added this function to sorting.c and checked it with exercise4.c.

2.5 Section 5

We had to define the following structure on times.h:

```
typedef struct time_aa {
   int N;                // Input size
   int n_elems;          // total number of elements to be averaged
   double time;        // average clock time
   double average_ob;   // avergae number of times the OB is executed
   int min_ob;           // minimum number of OB executions
   int max_ob;           // maximum number of OB executions
} TIME_AA, *PTIME_AA;
```

to store the execution times´ information.

Then we had to implement the following functions:

short average sorting time (pfunc_order method, int n perms, int N, PTIME_AA time):

> This routine fills a structure *PTIME_AA* with the information obtained from the sort function (*pfunc_order method* is a point to this routine), which orders the n_perms of size N that will be generated within the function.

*short generate sorting times (pfunc order method, char * file, int num min, int num max, int incr, int n perms).*

> This function executes the previous routine. The first number to be permuted is the minimum, and then we incremented by *incr* until we reach the maximum. Then it writes in the file the average time, maximum, minimum and average number of basic operations that the chosen sorting algorithm makes. The function returns OK or ERR.

*short save time table (char * file, TIME AA time aa, int n time).*

> This function is called by the previous one. It gets an array of structures TIME_AA, n_time is the number of elements of the array, and the file. It must print in each line N, the average sorting time, the average number of BOs, the maximum number of BOs and the minimum number of BOs of each element of the array of functions.

2.6 Section 6

We had to implement the function *int InsertSortInv (int * table, int ip, int iu),* that does the same as InsertSort but in reverse order (from highest to lowest), and to compare the results with InsertSort´s.

3 Tools and Methodology

In this practice, we both worked and coded in MacOS environment with the help of Atom. Since in MacOS we can execute and run in the terminal make commands as if it were Linux, we had no problem to check the exercises, but when we finished all the exercises, we had to use Linux to check valgrind errors. We just had memory leaks in Section 5 so we could easily fix them.

3.1 Section 1

In the first section we created a function that generates a random number between a minimum and a maximum (inf and sup), without using the module "%".

Firstly, we consulted the Chapter 7 (Random Numbers) of the book " Numerical recipes in C: the art of scientific computing ", to get an idea from where to start. We found an example which generated random numbers from 1 to 10, so we just had to generalize this example so that the 1 becomes the inferior limit and the 10 becomes superior limit. Then, we implemented it as shown in section 4.1.

The only difficulty we found was understanding why the denominator has to be RAND_MAX + 1 instead of just RAND_MAX. We got to the conclusion that it was because if we did not add 1 there was a really small possibility that rand() = RAND_MAX, in which case we would obtain a random number greater than the superior limit.

3.2 Section 2

For this section we implemented *int * generate perm (int N)* following the pseudocode, but including memory allocation and the convenient error control so that our code does not continue executing if any error pops up.

In order to implement the second loop, the one that makes the swaps between 2 elements of the array, we created an additional function *void swap (int *x, int *y)* that receives to pointers and changes its values.

At first we obtained repeated elements in the arrays when we executed our codes but we realised that it was trying to access elements that were not in the table by its indexes. We just had to fix how the for loops worked so that it did not access elements that were not in the array and it worked properly.

3.3 Section 3

In this section we took advantage of the function created in section 2 in order to create pointers to arrays of permutations and then store them in a array implemented as a double pointer, which is what is returned in the function *int ** generate permutations (int n perms, int N)*. This double pointer array will contain n_perms permutations/arrays of N elements each.

3.4 Section 4

We implemented the function *int InsertSort (int * table, int ip, int iu)* we had previously studied in class. So it was not hard to write the C code so that introduce as an input an array of integers and we receive back that array but sorting in increasing order.

3.5 Section 5

For this section we had to create three different functions:

short average sorting time (pfunc_order method, int n perms, int N, PTIME_AA time)

In this routine we had to understand how the clock worked, so we could measure the sorting times. We ended up setting the start of the clock before the loop in which the sorting algorithm was performed, and the end of the clock after this loop. We then subtracted the start value from the end value, divided it by the constant *CLOCKS_PER_SEC* and then by the number of permutations sorted to get the average time.

*short generate sorting times (pfunc order method, char * file, int num min, int num max, int incr, int n perms)*

*short save time table (char * file, TIME AA time aa, int n time)*

We had some problems when it came to implement the second and the third functions, since the last one received an array of structures, and we thought that the second one should only fill up a structure *TIME_AA*. We ended up creating an array of structures that for every iteration of the *AverageSortingTime* function, the values for one of the structures of the array were filled. At the end of the function, *SaveTimeTable* is called, and receives as an argument the array of structures, the number of elements of the array, and the file.

In the last function, we just had to print the elements of each structure, one structure per iteration.

We last struggled a little bit with valgrind in this section, since we did not realise we had to free the array of structures one by one. We implemented a for loop to free the elements of the array *array_times*, and then free the whole *array_times*.

3.6 Section 6

In this section, our goal was to implement the same function as in section 4 but instead of returning an array sorted in increasing order, *int InsertSortInv (int * table, int ip, int iu)* returns the array in decreasing order.

We just had to change a condition within a while loop. For InsertSort it looked something like this:

while (j >= ip && list[j] > a),

and for InsertSortInv:

while (j >= ip && list[j] < a)

Then we just had to store the same information as we did in section 5 but using this new sorting function, instead of InsertSort.

4. Source code

4.1 Section 1

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

4.2 Section 2

```c
void swap (int *x, int *y) {
  int aux;

  if (!x || !y) return;

  aux = *x;
  *x = *y;
  *y = aux;
}


int* generate_perm(int N)
{
  int *perm = NULL;
  int i;

  if (N<0) return NULL;

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


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

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

  return perm;
}
```

4.3 Section 3

```
int** generate_permutations(int n_perms, int N)
{
  int **perm, i;

  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);
  }

  return perm;
}
```

4.4 Section 4

```
int InsertSort(int* list, int ip, int iu)
{
  int i, j, a, count = 0;
  if (ip < 0 || iu < 0 || ip > iu) return ERR;

  for (i = ip; i <= iu; i++){
    a = list[i];
    j = i-1;
    while (j >= ip && list[j] > a){
      list[j+1] = list[j];
      count++;
      j--;
    }
    list[j+1] = a;
  }
  return count;
}
```

4.5 Section 5

```c
short average_sorting_time(pfunc_sort method,
                              int n_perms,
                              int N,
                              PTIME ptime)
{
  clock_t start, end;
  int i, j, res, min = 0,max = 0,sum = 0;
  int **perms = NULL;

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

  ptime->N = N;
  ptime->n_elems = n_perms;

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


  start = clock();
  for (i = 0; i < n_perms; i++){
    res = method(perms[i],0,N-1);
    if (res == ERR){
      for (j=0; j<n_perms; j++) free(perms[j]);
      free(perms);
      return ERR;
    }
    if (min == 0) min = res;
    else if (res < min) min = res;
    if (res > max) max = res;
    sum += res;
  }
  end = clock();

  ptime->time = ((double)(end - start) / (double)CLOCKS_PER_SEC) / (double)n_perms;
  ptime->min_ob = min;
  ptime->max_ob = max;
  ptime->average_ob = (double)sum/(double)n_perms;

  for (j=0; j<n_perms; j++) free(perms[j]);
  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, i_max;
  short result;
  PTIME array_times = NULL;

  i_max = (num_max - num_min)/incr + 1;
  array_times = (PTIME)malloc(i_max*sizeof(TIME));

  if(!array_times) return ERR;

  for (i = 0; i < i_max; i++){
    result = average_sorting_time(method, n_perms, num_min + incr*i, &array_times[i]);
    if (result == ERR){
      free(array_times);
      return ERR;
    }
  }

  store_time_table(file, array_times, i_max);

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

```c
short store_time_table(char* file, PTIME time, int n_times)
{
  int i = 0;
  FILE *f1 = NULL;

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

  f1 = fopen(file, "w");
  if (!f1) return ERR;

  for (i = 0; i<n_times; i++){
    fprintf (f1, "%i %f %f %i %i\n", time[i].N, time[i].time, time[i].average_ob, time[i].max_ob, time[i].min_ob);
  }

  fclose(f1);
  return OK;

}
```

4.6 Section 6

```
int InsertSortInv(int* list, int ip, int iu)
{
  int i, j, a, count = 0;
  if (ip < 0 || iu < 0 || ip > iu) return ERR;

  for (i = ip; i <= iu; i++){
    a = list[i];
    j = i-1;
    while (j >= ip && list[j] < a){
      list[j+1] = list[j];
      count++;
      j--;
    }
    list[j+1] = a;
  }
  return count;
}
```

5. Results, Plots

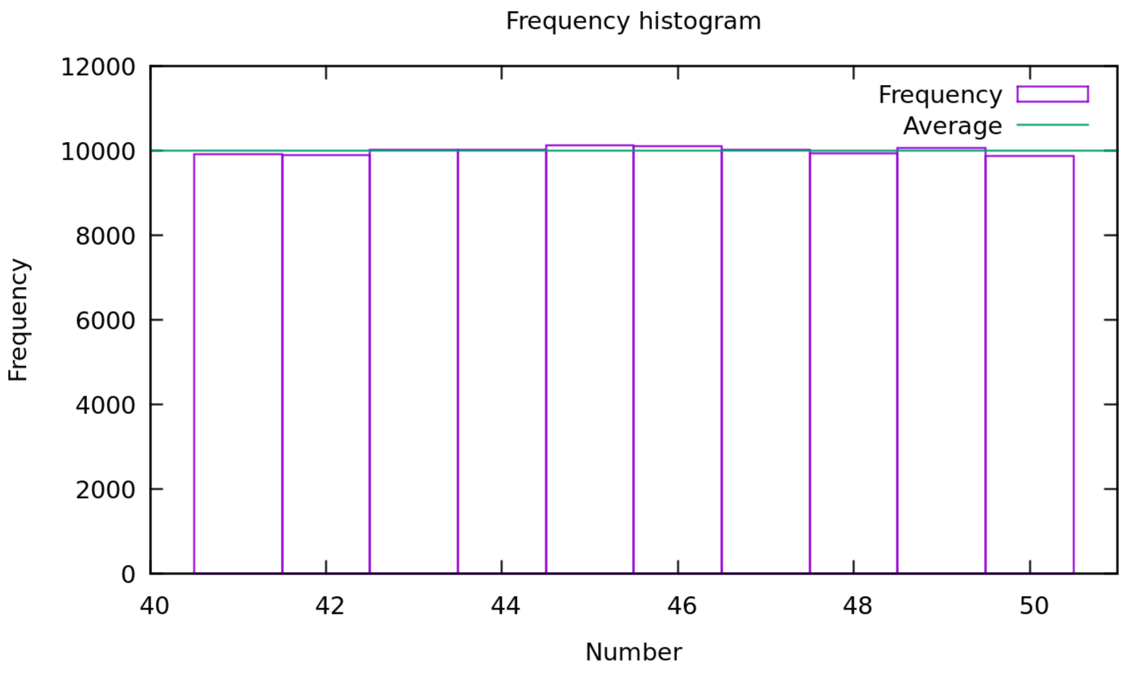
5.1 Section 1

Executing exercise1.c, we get a series of random numbers n, where inf <= n <= sup.

When running *./exercise1 -infLim 5 -supLim 10 -numN 7*, we get 7 random numbers between 5 and 10, such as: 5 8 9 5 7 7 6

To check if this program generates each number with the same probability, we generate 10.000 random numbers between 41 and 50 and count how many times each number has been generated and then we plot it with gnuplot.

As we can see in the histogram below, each number is generated approximately 1000 times, which means the probability of all of them been generated is equal.



5.2 Section 2

When we execute exercice2.c to check whether our code is correct or not, we obtain a series of permutations each of them with N elements and each of them sorted in a different way.

For instance, if we run this command: *./exercise2 -size 10 -numP 5*, we get 5 permutations of arrays of size 10 with numbers from 1 to 10 each of them. This would be the result:

```
1 4 8 10 3 7 6 5 2 9

3 5 1 9 6 4 2 8 7 10

10 9 5 1 4 2 3 7 8 6

10 5 7 6 4 9 2 8 3 1

4 2 8 1 9 6 10 7 5 3
```

5.3 Section 3

What we get when we run exercise3.c is the same result as when we run exercise2.c, we get some permutation with N elements each. The difference between these two exercises is that in exercise2, the main calls the function generate_perm several times, whereas in exercise3 generate_permutations is called only once and it is the own function the one which generates several permutations and the array with all the permutations is them printed, like this:

```
2 5 10 1 6 4 8 7 3 9

1 7 5 6 3 4 8 9 10 2

8 1 4 6 3 9 2 7 10 5

6 7 10 3 2 5 9 4 8 1

1 3 4 6 9 7 10 2 8 5
```

5.4 Section 4

We used the provided file exercise4.c to check if our code worked as it should. When we run *./exercise4 -size 10* we get an array that look like this:

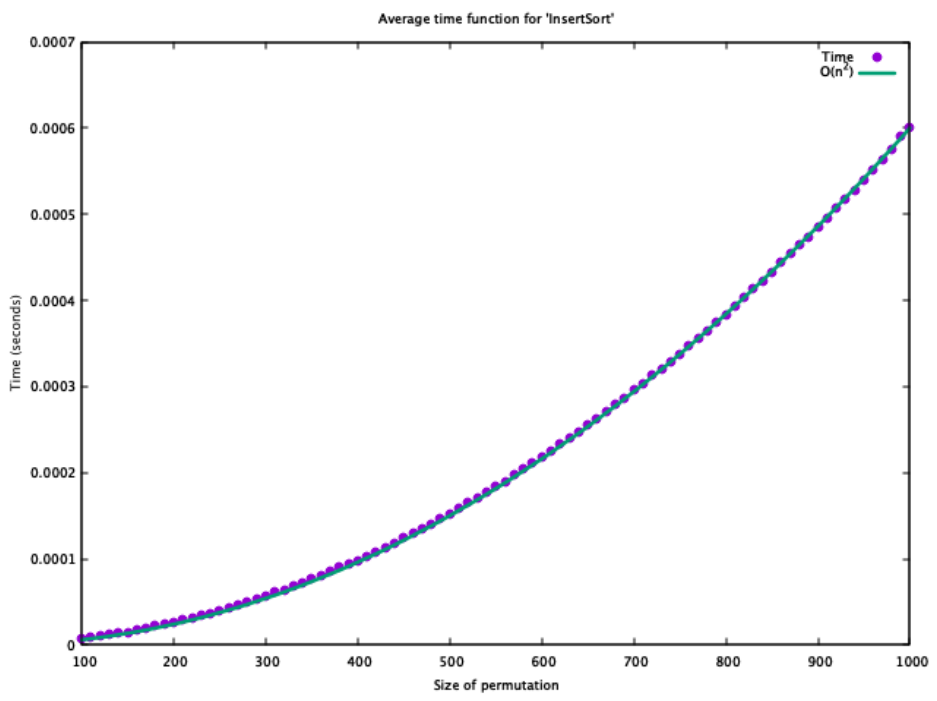
```
1     2 3     4 5 6     7 8 9 10
```

which is a sorted array with elements from 1 to 10 and correctly sorted in ascending order. Thus, we checked out code was correct.

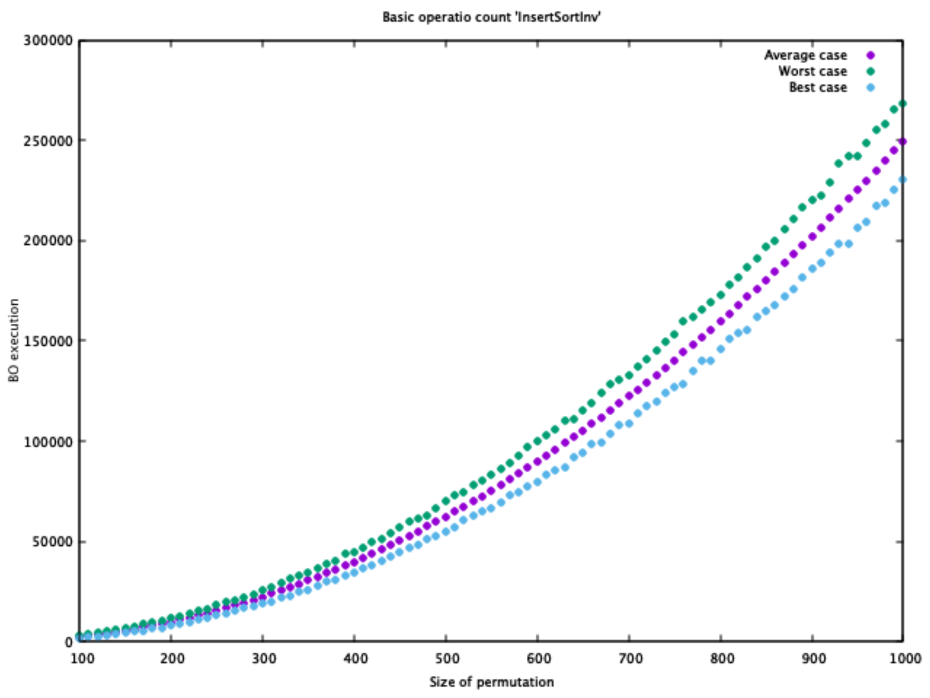
5.5 Section 5

In this section we couldn't check if out code was correct until we finished coding all three functions, so it took us longer to test it. Plus, it was the most difficult exercise of all to implement. Finally we check it with the help of exercise5.c and the command *./exercise5 -num_min 1 -num_max 5 -incr 1 -numP 5 -outputFile exercise5.log* that we found on the makefile. This generates permutation with sizes from 1 up to 5 and with and increment in size of 1, so we obtain 5 permutations.

To see that it worked properly we used bigger numbers and thus we would have a file that we could use later to generate the graphs showed below. This is what we ran: *./exercise5 -num_min 100 -num_max 1000 -incr 10 -numP 10000 -outputFile exercise5.log.*



In this plot we show the time that our algorithm took to sort in ascending order a total of 10000 arrays with sizes from 10 to 100. We can clearly see that it resembles a parabola, which is the same as O(n^2). We calculated the exact equation that fits this parabola formed by points and plotted it as well (green line).



This plot shows increment of BO executions while sorting the arrays. It shows the same results as what we have proven in class: the worst case, average case and best case for InsertSort are all O(n^2) and we can see that B <= A <= W.

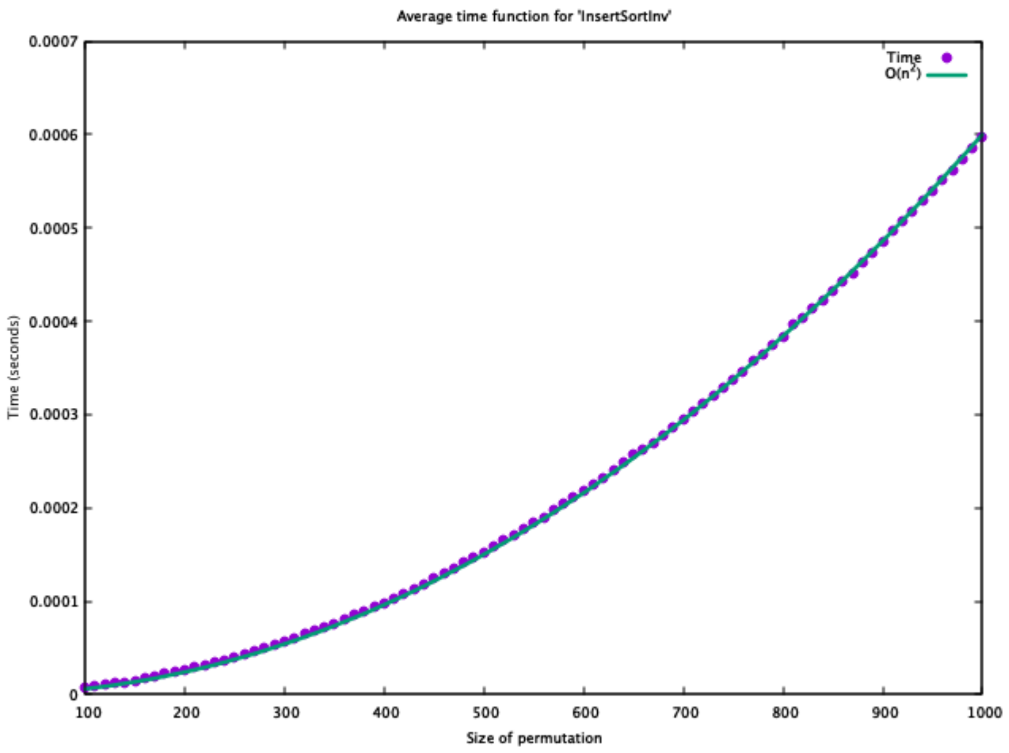
5.6 Section 6

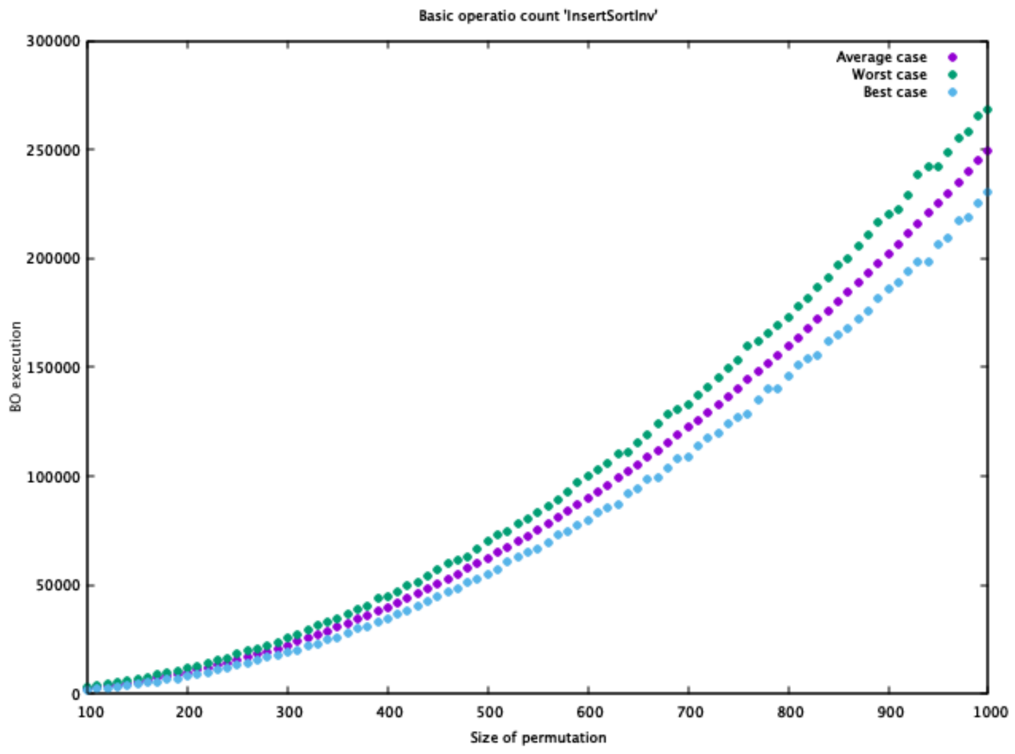
First, we just had to check if InsertSortInv function worked properly as InsertSort did.

We modified exercise4.c so that it calls InsertSortInv instead of InsertSort and check the array it returned was sorted in descending order, like this:

10 9 8 7 6 5 4 3 2 1.

Then we changed exercise5.c the same way we did with exercise4.c, so that all the arrays were sorted as the one showed previously all the information required to plot the following graphs were stored in a .log file.





These two graphs and he ones that illustrate InsertSort algorithm look very alike. In fact, the theoretical time of both algorithms is the same, as is the average, best and worst cases. That's why the plots for InsertSort and InsertSortInv are almost the same.

6. Answers to theoretical Questions.

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.**

   We consulted the book " Numerical recipes in C: the art of scientific computing ", more specifically chapter 7 (Random Numbers). We didn't find the algorithm itself but we found an example that helped us implement out random number generator, which behaves the following way:

   1. rand() generates a random number between 0 and RAND_MAX (a constant).
   2. When dividing rand() / RAND_MAX, we get a floating point number in the interval [0,1]. If we add 1 to the denominator, then the result would be in [0:1).
   3. We just have to multiply by the superior limit and add 1 to the number found so that this new result ends within [1,sup+1). The integers in this range are those going from 1 up to sup (both included).
   4. In case the inferior limit is not 1, we just have to sum the inf and adjust the multiplying factor this way: (sup - inf + 1).

An alternative to generate random numbers would be to count time and then operate with its digits, but this would require a higher coding level and the since the operations can be made in any way, it would be more difficult for another person to read and understand how it works.

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

InsertSort receives as an argument the list to be sorted, the first element, and the last one. It checks error and creates a counter to count the number of basic operations.

Then it executes the first loop, going through all the elements of the list, starting from the first, and ending in the last one. We set *j* as the previous element of the array, and calls the element to be compared *a*.

We jump into the inner loop, which checks that *a* is not the first element, and will be executing until the *a* is greater than the element in the position *j*, taking into account that *j* decreases by 1 each iteration, so we compare *a* with all the elements until we find the position of *a.* Within the loop we just move each iteration the element in position *j+1,* so as to make room for *a* when we find its position, and add one to the counter.

When we get out of the loop, we assign *a* its new position in the array, at the right of the smaller element, and return the number of basic operations.

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

The outer loop of InsertSort does not act on the first of the table, since this routine is based on the idea of creating a sorted table on the left part of the array, and then inserting one by one the other elements of the array. At the beginning, we do not have anything sorted, so we choose the first element of the array to start the sorted table. As it does not make any sense to compare an element with itself, we directly compare the second element with the sorted array (that in this case only consists on the first element).

In our implementation, the first element goes into the first loop, but we check on the inner loop that the element to be compared had to be greater than the first element:

Outer loop: *for (i = ip; i <= iu; i++);*

Inner loop: *while (j >= ip && list[j] > a));*

However, we could have just designed our outer loop to start directly with the second element: *for (i = ip+1; i <= iu; i++);*

4. What is the basic operation of InsertSort?

We have chosen as a basic operation of InsertSort *list[j] > a*, since it is a key comparison, it is located within the innermost loop it is a representative operation of the algorithm.

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

The worst case of InsertSort would be the array sorted the other way around, so when an element is inserted into the array, is going to be compared with all the elements: WA(N) = Ω (N^2/2 - N/2)

For calculating the best case, we assume that the array is already sorted, so that the inner loop is never triggered. In this case we would have a linear development, since the element that is going to be inserted into the sorted array, is only compared once. BA(N) = O(N)

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).

The graphs are almost the same. That means the execution time of sorting an algorithm using either InsertSort or InsertSortInv is theoretically equal, although it is obvious that in the practice it can vary slightly. This is because in the implementation of the algorithms there are no difference between both but the sign of the comparison, which doesn’t affect to the execution time. Therefore, the graphs showed for these algorithms are the same.

If f := graph InsertSort, g := graph InsertSortInv, we have f ~ g.

Lim f(N)/g(N) = 1

since, they both meet : f(N) = O(n^2) and g(N) = O(n^2)

7. Final Conclusions.

In this practice we have learnt and fully understood the theory studied in class in the first and second units of this course of analysis of algorithms.

But not only that, we believe that we have improve our skills while implementing certain functions or algorithms as well as for solving problems that may come up while coding. Some of these problems can be syntax errors, problems with pointers and double pointers but mainly with valgrind.

We wanted to highlight the fluency we have acquire when it comes to understanding and programming sorting algorithms, since we now understand their different methodologies and uses.

This assignment also puts into practice the complexity of the algorithms learnt during theory lessons, because we were able to see how the best, worst and average case of an algorithm work on a real environment. We confirmed that experimental values actually meet with the values we can theoretically calculate. This is represented in the graphics we created to illustrate the results. Besides, we have also learnt to plot them with gnuplot, a very simple and powerful tool that will help us for sure in the future.