Analisys of Algorithms
Escuela Politécnica Superior, UAM, 2019–2020

Set of Practices n. 1

Date of Delivert

- Thursday Groups: October 17th.
- Friday Groups: October 18th.

The delivery of source codes and documentation in electronic format corresponding to the practices of the subject AA will be carried out on the date indicated through the web page**https://moodle.uam.es/**.
Printed documentation will be delivered to the teacher at the beginning of the next class of practices.

Introduction

This first practice will serve as a settlement for future practices. It will be generated several .C files **permutations.c sorting.c times.c** that will be used as a library for the codes of the different C functions to be implemented; also it will be included in the .H files **permutations.h sorting.h times.h** the prototypes of the functions implemented in the .C files and the # **define**, such as:

```
#define ERR  -1
#define OK   (!(ERR))
```

It will be mandatory to conform to the given scheme of practices, that is, the prototypes of the functions should not be modified (except if indicated by the practice teacher).
Advices on the implementation of routines:

1. Check that the routine input arguments are correct. Independence of the functions.
2. Before leaving a routine or program, free all memory and close all files that are no longer used.
3. Check that the files can be opened, if not act as in point 2.
4. Check in each case that when reserving memory, it has been performed correctly and in case it could not have been reserved, before leaving the routine or the program proceed as in point 2.
5. Check programs in extreme conditions to verify good program behavior (e.g. very large tables, entries illegal ...)
6. Use runtime environments in protected mode (Windows XP, Vista, Windows 7, Windows 8, Windows 10, Unix, Linux, OSX/iOS ... Never bare DOS).
7. **The practices will be corrected in a Unix system or Linux. The programs must be developed in ANSI C so that be portable.**

The file **exercise1.zip** with the C templates is available on the Moodle platform which include the prototypes of the functions required throughout this practice as well as test programs for each section.
The practice will begin by developing in the file **permutations.c** a set of C routines that generate random permutations of a certain number of elements. These routines will be used later to obtain the sorting algorithm inputs. Next, a known sorting algorithm will be implemented in the file **sorting.c** and finally routines that will measure the execution times of the sorting algorithm implemented using the permutations generated will be implemented in the file **times.c**. Each practice will be carried out in a period of four to five weeks.

First Section

Generation of permutations

1. The C **rand** routine in the library **stdlib** generates equiprobable random numbers between 0 and the value of **RAND_MAX**. Use this function to build a routine **int random_num (int inf, int sup)** that generates equiprobable random numbers between the integers **inf, sup**, both inclusive. Implement this routine in the file **permutations.c**.

   Use the file **exercise1.c** to check the correct functioning of the routine you just implemented.

   Check by developing a histogram if the generation function is equiprobable for each digit.

   Obs.: The level of randomness of the function implemented will be valued, so it is recommended to think carefully about your design instead of implementing the first idea you have. As help it is recommended to consult the Chapter 7 (Random Numbers) of the book " Numerical recipes in C: the art of scientific computing " of which several copies are available in the library.

2. The following pseudocode provides a method of generation of random permutations

```
for i from 1 to N:
   perm[i] = i;

for i from 1 to N:
   swap perm[i] with perm[random_num(i, N)];
```

   where **random_num** is the routine developed in the previous exercise.

   Implement a routine **int * generate_perm (int N)** in the C file **permutations.c** for that algorithm.

   Use the file **exercise2.c** to check the correct functioning of the routine you just implemented.

Segundo bloque

3. Insert the routine **int ** generate_permutations (int n_perms, int N)** into the file **permutations.c** which generates **n_perms** equiprobable permutations of **N** elements each by using the function **generate_perm** from the previous exercise.

   Use the file **exercise3.c** to check the correct functioning of the routine that you just implemented.

Sorting Algorithms

In this section the average clock execution time will be determined experimentally So it will be the average, best and worst number of times the Basic Operation (OB) is executed for the insertion algorithm *InsertSort* on tables of different sizes, finally the result will be compared with the theoretical analysis .

4. Implement in the file **sorting.c** a function **int InsertSort (int * table, int ip, int iu)** for the insertion method *InsertSort*, this function returns ERR in case of error or the number of times the OB has been executed in case the table is ordered correctly, **table** is the table to sort, **ip** is the first element of The table and **iu** is the last element of the table.

   Use the file **exercise4.c** to check the correct functioning of the routine you just implemented.

Third Section

Execution times

5. Define in **times.h** the following structure that will be used to store the execution times of an algorithm on a set of permutations:

```
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;
```

   Implement the function

   short average_sorting_time (pfunc_order method, int n_perms, int N, PTIME_AA time),

   in the file **times.c** where **pfunc_order** is a pointer to the sort function, defined as:

   typedef int (* pfunc_order) (int *, int, int);

   this typedef should be included in `sorting.h`, **n_perms** represents the number of permutations to be generated and sorted by the used method (in this case insertion method), **N** is the size of each permutation and **time** is a pointer to a type structure **TIME_AA** that at the exit of the function will contain:

   - the number of permutations averaged in the **n_elems** field,
   - the size of the permutations in the **N** field,
   - the average execution time (in seconds) in the **time** field,
   - the average number of times the OB was executed in the **average_ob** field,
   - the minimum number of times the OB was executed in the **min_ob** field
   - the maximum number of times the OB was executed in the **max_ob** field

   for the sorting algorithm sort **sort_method** on the permutations generated by the function **generate_permutations**.

   The routine **average_sorting_time** returns returns ERR in case of error and OK in case the tables are ordered correctly.

   Also implement the function:

   short generate_sorting_times (pfunc_order method, char * file,
   int num_min, int num_max, int incr, int n_perms)

   which writes in the file **file** the average clock time, and the average, minimum and maximum number of times that the OB is executed in the execution of the sorting algorithm **method** with size permutations in the range from **num_min** to **num_max**, both included, using increments in size **incr** and **n_perms** permutations for each size. The routine will return the ERR value in case of error and OK in opposite case.

   The funcion **generate_sorting_times** calls a function

   short save_time_table (char * file, TIME_AA time_aa, int n_time)

   which will save in the file **file** a table with five columns corresponding to size **N**, average clock tome **time** average **average_ob**, maximum **max_ob** and minimum **min_ob** of times the OB is executed; the array **time_aa** contains the time data and **n_times** is the number of array elements.

   Use the file **exercise5.c** to measure the execution times of the algorithm **InsertSort**

   Comment: It would not be strange if you run your program you get that the execution time is zero, that is because the processor speed is so high that it does not even consume a Clock TIC measured by the **clock** function during the call to **average_sorting_time**. Then, to measure the runtime it will be necessary to introduce some delay mechanism or use another more precise function for time measurement.

   As a suggestion, the best option to avoid the previous situation is to choose a a larger number of tables to average or choose larger table sizes.

Another option is to use more precise functions than the function of Standard library **clock** for the time measurement. By example on UNIX and LINUX there is the system function **clock_gettime** which gives an accuracy of nanoseconds $(10-9)$. This function is not defined in ANSI C and does not exist in Windows (although there are calls equivalent) or OSX.

6. Sometimes it is interesting to sort a table in values ??from highest to lowest. Implement a routine **int InsertSortInv (int * table, int ip, int iu)** with the same arguments and return as **InsertSort** but sorting the table in reverse order (from highest to lowest), obtain the execution times and compare the results obtained with those of the **InsertSort** routine

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.
2. Justify as formally as you can the correction (or put another way, why order well) of the algorithm **InsertSort**.
3. Why does the outer loop of **InsertSort** not act on the first element of the table?
4. ? What is the basic operation of **InsertSort**?
5. Give execution times based on the input size n for the worst case $W_{BS}(n)$ and the best case $B_{BS}(n)$ of **InsertSort**. Use asymptotic notation $(O, \Theta, or, \Omega$, etc.) whenever possible.
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).

Material to be delivered in each of the sections

Documentation: The documentation will consist of the following sections:

1. **Introduction:** It consists of a technical description of the work to be carried out, what are the objectives they intend to reach, what input data is required and what data is obtained from the output, so like any kind of comment about the practice.
2. **Printed Code:** The routine code according to the corresponding section. The code also includes the header of the routine.
3. **Results:** Description of the results obtained, comparative graphics of the results obtained with the theoretical ones and comments on them.
4. **Questions:** Answers to theoretical questions.

The printed documentation will be given to the practice teacher in the class corresponding to the day after delivery of the practice. On the cover should be included The names of the students. All the files necessary to compile the practice and documentation have to be stored in a single compressed file, in zip format, or tgz (tgz represents a tar file compressed with gzip). The name of that file will be **name1 last_name1 name2 last_name2.zip** or **name1 last_name1 name2 last_name2.tgz**. Where **name last_name** is the name and last name of each of the members of the couple.
Additionally, the practices should be stored in some storage medium. (pendrive, CD or DVD, hard disk, remote virtual disk, etc.) by the student for the day of the practice exam in December.

Warning: The importance of carrying a pendrive is stressed **in addition to other storage media such as usb disks, cd, remote virtual disk, email to own address, etc**, since it is not guaranteed that they can be mounted and accessed each and every one of them during the exam, which will mean the grade not pass s practices.

In order to normalize the compilation and execution of the various programs, it is suggested use the make tool next to the Makefile file included in the file **exercise1.zip**. In this Makefile file the options exercise1, exercise2, ..., exercise5 will be implemented that compile the programs of the different sections. The exercise1_test, exercise2_test, ..., exercise5_test options will also be implemented to run the programs. You can also use development environments such as Netbeans, Anjuta, Visual Studio, etc.

Instructions for the delivery of the practice codes

The delivery of the source codes corresponding to the practices of the subject AA will be carried out through the Web page **https://moodle.uam.es/**.