**Assignment 1**

**Draghici Bogdan**

**The Faculty of ACE, Craiova**

**First Year, Calculatoare Romana**

**CR 1.2.A**

**Problem 1**

**I. Problem statement**

Let us consider a book with numbered pages. The numbering requires the use of n digits. Design an algorithm to determine the numberof book pages.

For example, for a book with 500 pages the number of digits is:

9 (1...9) + 2 X 90 (10...99) + 3 X 401 (100...500) = 9 + 180 + 1203 = 1392 digits. Given input 1392 you must print output 500.

**II. Pseudocode description of all the algorithms used in the lab homework**

The main algorithm used to solve the problem is contained in the next function:

1. *function no\_pages (no\_digits)*






9. *write "with ‘no\_digits’ digits can be created ‘pages’ pages, remaining ‘’ digits unused”*

This function uses the variable i to count the number of digits of the pages we can generate, in sum is added all the digits that we used to generate the pages, in pages we find how many we can create with the given number of digits no\_digits, counter represents the number of pages that have i number of digits.

On line 2, I initialized:

* i=1, representing the pages that are the first to be created (1digit pages)
* sum=0, because we did not use any digits to generate pages so far
* pages=0, because we have not generated any pages yet
* counter=9, representing the number of numbers that have only 1 digit

On line 3, I use a repetitive structure to verify if I can generate all the pages that have maximum i digits by comparing the sum between sum of digits of all pages, with a smaller number of digits than i, that have already been generated, and the number of digits needed to create all the pages with i digits.

While this condition is true, on line 4 it is added to sum the number of digits of all the pages that have i digits, represented by the number , on line 5 it is added to pages the number of pages with i digits, which is exactly the variable counter, on line 6 we increment the number of digits of the pages we may be able to fully generate and on line 7 we multiply counter with 10, so that it will represent the number of pages that have i digits.

When we are no longer able to generate all the pages with i number of digits, the repetitive structure will end. After this happens, for the current i we will have the following values:

* in sum:
* in counter:
* in pages:

Next, on line 8 we add to pages the rest of the pages that we can generate with the remaining digits, those being represented by the difference between no\_digits and sum divided by the number of digits i of the current pages.

On line 9 is written the number of pages that can be created, and the remaining digits that could not form another page.

**III. Algorithms' memory and computational complexity**

The algorithm described above has an execution time and O(1) memory, because is using simple variables.

At initial step we have

For each subsequent step we have , till

…………………………….

e.g.1:

For n= 2376, the algorithm stops at because , meaning it only did 2 steps and verified the third one. After exiting the repetitive structure, it will divide the difference between n and the sum from the last step, , with the current step, 3: , giving the result 729+99=828.

e.g.2:

For n=6473, the algorithm stops at , because , meaning it only did 3 steps and verified the fourth one. After exiting the repetitive structure, it will divide the difference between n and the sum from the last step, , with the current step, 4: , giving the result 896+999=1895.

The number of steps it makes is smaller than , because and .

The complexity of the algorithm is , and the memory used is O(1).

1. **Best case scenario**

In the best case scenario, this algorithm will only verify the condition, but not enter the repetitive structure, making 3 elemental times for the verification. This can only happen for n < 9. The function will also make the first 4 attributions, the adding to the variable pages that has 4 execution time, and the writing that has 5 elemental execution time.

The algorithm has elemental execution time.

1. **Worst case scenario**

In the worst case scenario, the algorithm will enter the repetitive structure for the maximum number of times. In case n is an int variable, it will enter a maximum 9 of times (for ). The number of operations that are made inside the while is: 3 for attribution of sum and 2 for other attribution. At every step it does 3+2\*3=9 elemental execution time. Will also verify the condition of while 10 times. In the end, we will have elemental execution time.

**IV. Experimental data**

For the generation of the input data sets I used the rand() function from the library math.h, and also an function implemented by myself, that generates a random number from a given range.

1. *function random(min, max)*

RAND\_MAX is a constant which represents the biggest value that can be generated using the function rand(), and it’s value is 32767.

On line 2 I initialized counter with the length of the range and the variable in which the number will be created with 0.

On line 3 is a repetitive structure that will be executed for times, in order to obtain a number in the range.

On line 4, the number is multiplied by RAND\_MAX and also added a new random value generated by the rand() function. On line 5 the counter is divided by the maximum value that can be taken by a number generated in rand().

So, when the number is multiplied by RAND\_MAX, the counter is divided by RAND\_MAX, in order to get to the given range as quickly as possible.

After the repetitive structure ends, the number created might exceed the range length, so we must do the rest of the division with the range length+1, then add the first value of the range in order to obtain a number between min and max.

e.g. For range 1,000- 1,000,000: counter=999,000

We enter the repetitive structure the number gets a random value, lets say 5684. Counter =. At second repetition, the number generated is 23091, so number= and then counter=0. In the end, the number generated is: = = 457,533.

**V. Results and Conclusion**

The algorithm runs too fast, giving time null on every input, so I had to put a for with 5000000 steps in order to obtain a time different from 0. In the next graphics, it can be observed the inconsistence of time on different data inputs.

Next is the graphic of average data time:

1. **Summary of achievements**

My biggest achievement and the one I am most proud of, is that I had the occasion to optimize an already made algorithm from a previous homework of AD course.

1. **Brief description of the most challenging/ interesting achievements**

The most challenging achievement was finding a way to optimize the algorithm in O (log n) time.

1. **Future directions for extending the lab homework**

I think that there might be an even faster way to solve this problem, and that maybe I will figure it out in the future.