Profiler: A useful library for the Fundamental Algorithms laboratory

During the Fundamental Algorithms lab, you will often need to:

- Generate random data for your algorithms
- Measure thier performance (number of operations, execution time)
- Draw charts

In order to ease your work, the Profiler.h library provides a series of helpful functions .

Generating Random Arrays

The function *FillRandomArray* will fill an array with random values in a given range. Optionally, the generated array can be sorted (ascending or descending) or it can have unique elements.

```
template <typename T>
void FillRandomArray(
        T *arr,
        int size,
        T range_min=10,
        T range_max=50000,
        bool unique = false,
        int sorted=0
);
```

You don't need to understand what a template is. Just replace T with any type you want (int, char, double, float).
The parameters are:

- **arr** – the array that must be filled with random values;
- **size** – the size of the array above;
- **range_min**, **range_max** – the bounds for the range of possible values. They must have the same type as the array arr (for example, if you need double values from 2 to 5, you will write 2.0 and 5.0). If the unique parameter is true, the length of the interval must be bigger than the size of the array (except for double and float types). These parameters are optional. If not specified, the default values will be used.
- **unique** – if set to true, the generated random values will be unique. This parameter is optional. If not specifies, the default value (false) will be used.
- **sorted** – possible values are 0, 1 and 2. The value 0 means that the elements will have a random order, 1 – they will be sorted in ascending order and 2 – they will be sorted in descending order. The default value is 0, if this optional parameter is not specified.

Another function that one might want to use is *IsSorted*:

```
template <typename T> bool IsSorted(T *arr, int size);
```

This function receives an array and its size and returns true if the array is sorted in ascending order or false otherwise.

Measuring performance

The Profiler class will help you count the number of operations your algorithms perform, measure the execution time, then draw tables and charts with these measurements.
First of all, we need a profiler object, that can be declared as a global variable at the begining of your code:

```
Profiler profiler("demo");
```

Your profiler can have a name that will be used in the report that will be generated.

For counting operations, we will use the *countOperation* function:

```
__forceinline void countOperation(const char *name, int size, int increment=1);
```

The parameters for this method are:

- **name** – the name of the operation to be measured. Legal characters for a name are lowercase and uppercase letters, digits, hyphen ('-'), underscore ('_') and dot ('.')
- **size** – the data size for the running algorithm. For example, if you are sorting an array, you should use its length. This size will be the x-axis on your plots.
- **increment** – sometimes, one might want to count several operations of the same type at once. Specifying an increment will avoid calling this function repeatedly. This parameter is optional and the default value is 1.

The following example shows us how to compute multiplications when raising a number x to the power n (the naive version):

```
int slow_pow(int x, int n){
	int p = 1;
	int i;
	for(i=0; i<n; ++i){
		//count the multiplications
		profiler.countOperation("multiplication", n);
		p *= x;
	}
	return p;
}
```

Similarly to counting operations, we can measure the execution time of a fragment of code. The methods for doing this are:

```
__forceinline void startTimer(const char *name, int size);
__forceinline void stopTimer(const char *name, int size);
```

The parameters are similar with the ones from *countOperation* and the methods must be called in pairs:

```
profiler.startTimer("your-operation", n);
//here goes the code you need the execution time for
...
profiler.stopTimer("your-operation", n);
```

Finally, when you want to display your report, all you need to do is call

```
profiler.showReport();
```

A web page containing the tables and charts will be created and probably opened for you.

Mixing counters and timers

Sometimes, you want to count the number of operations and measure the execution time for the same function. Unfortunatelly, as in qunatum mechanics, you can't measure both at the same time. The reason is that the method countOperation performs some operations itself and sometimes these extra-operations will take more time than the operations you actually need to measure.

For this reason, if you need to measure time for a function that has calls to *countOperation*, you should disable these calls first, by calling:

```
profiler.disableCounters();
```

You can enable them back, by calling

```
profiler.enableCounters();
```

It is also not advised to nest two timers, as the values measured by the outer one will not be accurate.

Grouping the measurements

When you need to compare two or more algorithms, you will want to show them in the same table and in the same chart.

The profiler provides the method createGroup:

```
void createGroup(const char *groupName, const char *member1, const char *member2, ...);
```

The first parameter must be the name of the newly created group, followed by a variable number of members (up to 10), that will be grouped together.

Other scenarios might involve adding two series (such as the number of assignments and the number of comparisons), in order to build a new one. The method for this is:

```
void addSeries(const char *newName, const char *series1, const char *series2);
```

The parameteres are the same as in the previous method, except that you can only add two series.

!!!! You can find a working example – **test.cpp**, in the archive available at:
http://users.utcluj.ro/~cameliav/fa/profiler.zip