PRG (E.T.S. d'Enginyeria Informàtica)
Academic Year 2019-2020

Lab activity 3 *(2 sessions)*
Empirical Measurement of Computational Complexity

Departament de Sistemes Informàtics i Computació
Universitat Politècnica de València

Contents

1 Context and previous work

This lab activity corresponds to *"Unit 2. Analysis of Algorithms, Efficiency and Sorting."* The objectives are the following:

- To introduce the empirical analysis of algorithms in a real environment.
- To graphically represent the growth rate of the temporal cost obtained empirically in order to confirm theoretical results.
- To infer approximated functions that define the time behaviour of an algorithm.
- To use the empirical results for comparisons and predictions.

Before the lab session, students must read the lab report and try to solve (as far as possible) the proposed problems. The activities will be developed during 2 lab sessions.

2 Measuring a classical problem: the linear search

A complete empirical analysis problem is presented in this section. The problem is the linear search, i.e., the search for an element in a linear array that may not be ordered.

2.1 Problem statement

Given an array of elements (a) of a certain set (e.g., integer numbers) and given an arbitrary element (e) of the set (e.g., a specific integer number), the linear search problem consists of seeking the first position of the array a that stores the element e. When e is not present in a, an invalid index (e.g., -1) is given as a result to signal the absence of the element. This problem can be solved by the following algorithm:

```
Input: array $a$ of $n$ elements, element $e$
Output: the index $i$ such that $a[i]$ is $e$, or -1 when $e$ is not in $a$
Process:
    From position 0 to $n-1$
        If element in current position of $a$ is equal to $e$
            Return current position
    Return -1
```

A possible Java implementation for that algorithm for arrays of integers is the following method:

```java
public static int linearSearch( int [] a, int e )
{
    int i=0;

    while( (i < a.length) && (a[i] != e) ) i++;

    return (i<a.length) ? i : -1;
}
```

2.2 Case analysis

When we face to the analysis of an algorithm, the first parameter we must define is the *size* of the problem. In this case, the size of the problem is clearly the size of the array, since it determines the number of iterations of the `while` loop (by the condition `i<a.length`).

Apart from that, we have to study whether the algorithm presents *significant instances* or not. The linear search problem presents significant instances. We are going to focus our attention to *best and worst cases* in order to simplify the analysis. Those cases are defined by the second condition of the `while` loop (`a[i]!=e`):

Best case: when the element `e` is in the first position of `a` (i.e., `a[0]==e`) because no more iterations are required.

Worst case: when the element `e` is not present in `a`, since all the array must be explored to discover that the element is not present.

With this previous analysis, we can state that the best case is constant and the worst case is linear. Consequently, using n =`a.length` as the input size of the problem, the asymptotic bounds for the algorithm are $\Omega(1)$ and $O(n)$, lower and upper bounds respectively. Then, we can say that the temporal cost function $T(n)$ belongs to the intersection of two sets: $T(n) \in \Omega(1) \cap O(n)$, where $\Omega(1)$ is the set temporal functions whose lower bound is constant, and $O(n)$ is the set of temporal functions whose upper bound is linear.

The average case is difficult to be calculated. Some simplifications can be stated to simplify the calculations for the average case. For example, we can state that the element to be searched is always present in the array and that the probability of finding the element in any position is the same. In this case the final bound for the average case gives us that $T^{\mu}(n) \in \Theta(n)$.

2.3 The structure of a measurement experiment

The empirical analysis should be performed after the theoretical one. The following points should be taken into account during the design of the empirical analysis:

- **The time measurement must be done for several sizes**: the objective is to obtain a cost function whose parameter is the size of the problem; several sizes must be used to get the profile of the function.
- **Significant instances must be measured separately**: best, worst, and average cases usually present different growth rates, and consequently different cost functions; thus, they must be measured in different parts of the code.
- **Several measures must be taken to get significant results**: a unique measurement for each value of the input size is not significant since it can be affected by conditions of the environment (e.g., other processes are running in the computer). Therefore, in order to guarantee a correct result several time measures must be taken and the average time must be considered.

The time measurement of the algorithm can be presented as the process:

1. Read the current system time and store it in t_I (initial time)
2. Execute the algorithm (method)
3. Read the current system time and store it in t_F (final time)
4. The difference between t_F and t_I is the time the algorithm used to solve the problem

This process can be done by using an external clock, but it is more precise to use the internal clock. Java provides the method

```
static long nanoTime()
```

in `java.lang.System`, that returns the current value of the most precise available system timer, in nanoseconds (although resolution may be lower, but at least presents a milliseconds resolution). Therefore, the usual Java code for time measurement looks like:

```
long initialTime, finalTime, elapsedTime;

initialTime = System.nanoTime();
// Call to the method
finalTime = System.nanoTime();
elapsedTime = finalTime - initialTime;
```

where `elapsedTime` will store the time the method employed to solve the problem. This measure is done many times and the average time is calculated. However, for extremely fast cases (e.g., best case of the linear search), it is usual to include the repetition loop inside the code to be measured. In this case, the loop overload is taken as inappreciable.

Finally, time measures must be properly presented. The usual form is a table that shows in each row the size of the problem and the measured times for the corresponding instances. A typical view of this table is:

linearSearch.out

```
# Size           Best         Worst        Average   (ms)
#-------------------------------------------------------------
  100000         0.11         55.08         23.95
  200000         0.06         95.38         46.76
  300000         0.04        147.48         73.20
  400000         0.05        215.72        100.43
  500000         0.04        336.80        138.77
  600000         0.04        427.71        188.74
  ......
```

Taking into account all these factors, a possible implementation of the empirical analysis is provided in PoliformaT. The code `MeasurableAlgorithms.java` includes, among others, the method `linearSearch( int [], int )` to be analysed. The program class `MeasuringLinearSearch.java` implements the code that performs the time analysis for the different significant instances of that algorithm.

Activity 1: creating the package `pract3` in the *BlueJ* project `prg`

Open in *BlueJ* the project (`prg`) and create a new package `pract3` including the classes `MeasurableAlgorithms.java` and `MeasuringLinearSearch.java`. These Java files are available in `PoliformaT` of `PRG`.

Activity 2: getting the running times for the Linear Search algorithm

Run the program class `MeasuringLinearSearch` for getting the table of running times for different sizes and save it in a file. It is recommended, to avoid overload, to run it from the command line. For instance, to save the results in the file `linear_search.out`, you could run from the command line in the `$HOME/DiscoW/prg` folder:

```
$ java -cp . pract3.MeasuringLinearSearch  > pract3/linear_search.out
```

It is possible that the JVM complains because compiler version differs from that of *BlueJ*. In that case, you must recompile the code by running from the command line in the `$HOME/DiscoW/prg` folder the following:

```
$ javac pract3/MeasurableAlgorithms.java
$ javac pract3/MeasuringLinearSearch.java
```

3 Graphical representation

Numerical results are usually best interpreted when its graphical representation is available. In this section we show how to use the *gnuplot* tool to obtain a graphical representation of the output results and to obtain approximated functions to the empirical results, which can be used to properly compare the algorithms and to obtain predictions.

3.1 Gnuplot

Gnuplot is a command-line based tool that allows to graphically represent mathematical functions and data points in two and three dimensions. The tool is started by writing `gnuplot` at the command line (terminal). The aspect it presents is similar to the following:

Gnuplot

```

    G N U P L O T
    Version 4.6 patchlevel 0    last modified 2012-03-04
    Build System: Linux x86_64

    Copyright (C) 1986-1993, 1998, 2004, 2007-2012
    Thomas Williams, Colin Kelley and many others

    gnuplot home:     http://www.gnuplot.info
    faq, bugs, etc:   type "help FAQ"
    immediate help:   type "help"  (plot window: hit 'h')

Terminal type set to 'wxt'
```

Gnuplot accepts commands with modifiers. In our case, the most important commands are the following:

- **plot**: to plot datafiles or built-in functions; the most important modifiers are:
  - *datafile*: it is specified between quotes and says where is the data to be plotted; lines starting with `#` are ignored
  - **title** *string*: it specifies the name to be given to the datapoints (legend)
  - **using** *i:j*: it specifies which columns of the datafile are going to be used (i for the X axis and j for the Y axis)
  - **with** *format*: it specifies the format for plotting (usual values for *format* are `lines`, `points` and `linespoints`)

  Example: `plot "linearSearch.out" using 1:3 title "Worst" with lines`
- **replot**: the same meaning and modifiers as *plot*, but it does not clean the graphical view and allows to see many graphics at once; *replot* alone allows to redraw the graphical view
- **set xrange** *[begin:end]*, **set yrange** *[begin:end]*: sets the range of size X or Y axis between *begin* and *end* values
- **set xtics** *interval*, **set ytics** *interval*: sets the interval between the marks in the X or Y axis
- **set xlabel** *string*, **set ylabel** *strings*: sets the labels for X or Y axis
- **load** *file*: loads a text file with *gnuplot* commands that are executed by *gnuplot*
- **fit**: allows to fit a predefined function with some free parameters to a set of datapoints

  Example

  ```
  f(x)=a*x
  fit f(x) "linearSearch.out" using 1:3 via a
  ```

  example of output when running the fit command

  ```


   Iteration 0
   WSSR        : 3.84408e+12       delta(WSSR)/WSSR   : 0
   delta(WSSR) : 0                 limit for stopping : 1e-05
   lambda      : 620484

  initial set of free parameter values

  a               = 1
  /

   Iteration 1
   WSSR        : 3.17693e+10       delta(WSSR)/WSSR   : -120
   delta(WSSR) : -3.81231e+12      limit for stopping : 1e-05
   lambda      : 62048.4
  ```

```
resultant parameter values

a               = 0.0916081
/

 Iteration 2
 WSSR        : 60705             delta(WSSR)/WSSR   : -523338
 delta(WSSR) : -3.17692e+10      limit for stopping : 1e-05
 lambda      : 6204.84

resultant parameter values

a               = 0.000859616
/

 Iteration 3
 WSSR        : 28999.2           delta(WSSR)/WSSR   : -1.09334
 delta(WSSR) : -31705.8          limit for stopping : 1e-05
 lambda      : 620.484

resultant parameter values

a               = 0.000768868
/

 Iteration 4
 WSSR        : 28999.2           delta(WSSR)/WSSR   : -1.09331e-10
 delta(WSSR) : -3.17051e-06      limit for stopping : 1e-05
 lambda      : 62.0484

resultant parameter values

a               = 0.000768868

After 4 iterations the fit converged.
final sum of squares of residuals : 28999.2
rel. change during last iteration : -1.09331e-10

degrees of freedom    (FIT_NDF)                        : 9
rms of residuals      (FIT_STDFIT) = sqrt(WSSR/ndf)    : 56.7638
variance of residuals (reduced chisquare) = WSSR/ndf   : 3222.13

Final set of parameters            Asymptotic Standard Error
=======================            ==========================

a               = 0.000768868      +/- 2.893e-05    (3.763%)


correlation matrix of the fit parameters:

               a
a              1.000
```

This means that function `f(x)=a*x` is fitted by varying the free parameter `a` to the data points of columns 1 and 3 of the file `linearSearch.out`. The best fit for the example used is the value of `a` at the end of the output in the section `Final set of parameters`.

- **print** $f(x)$: prints the value of a predefined function for a given value of x

Activity 3: representing empirical results

In order to represent by using *gnuplot* the empirical results of running the version of the linear search algorithm we provided to you, you have to execute in the command line the corresponding class and send the output to a file, as shown before in the Activity 2:

```
java -cp . pract3.MeasuringLinearSearch > pract3/linearSearch.out
```

You have to prepare the file `linearSearch1.plot` with the following contents:

linearSearch1.plot

```
set xrange [0:110000]
set yrange [-20:]
set xtics 200000
set ytics 100
set xlabel "Size"
set ylabel "Microseconds"
set key left
set grid

plot "linearSearch.out" using 1:2 title "Best case" with points, \
     "linearSearch.out" using 1:3 title "Worst case" with points, \
     "linearSearch.out" using 1:4 title "Average case" with points
```

then, you have to start *gnuplot* and type in the following command when the prompt appears:

```
gnuplot> load "linearSearch1.plot"
```

The image that is shown must be similar to the one presented in Figure 1.

If you want to save the output in a `.pdf` file, you have to use the following file:

linearSearch2.plot

```
set xrange [0:110000]
set yrange [-20:]
set xtics 200000
set ytics 100
set xlabel "Size"
set ylabel "Microseconds"
set key left
set grid
```

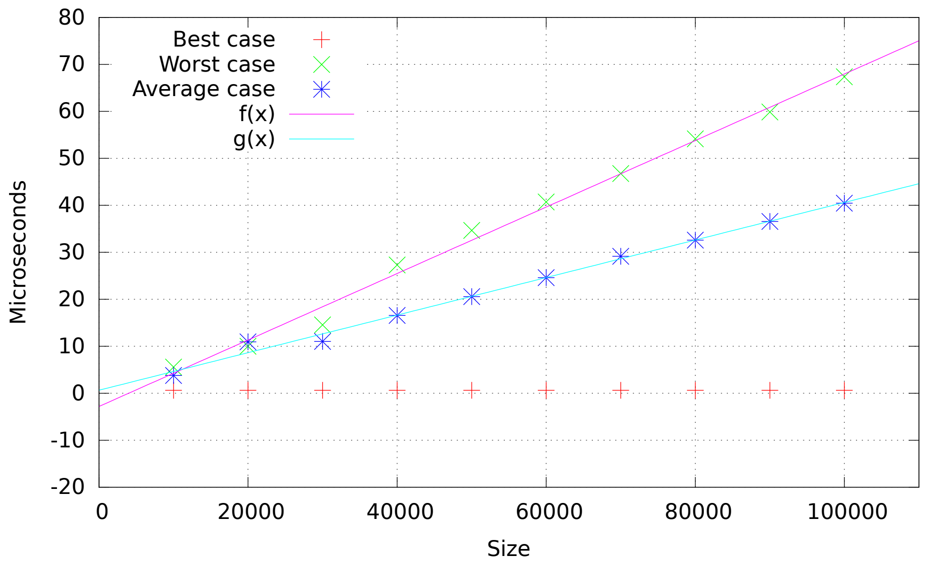

Figure 1: Linear search results by *gnuplot*

```
set term pdf colour enhanced solid
set output "linearSearch.pdf"

plot "linearSearch.out" using 1:2 title "Best case" with points, \
     "linearSearch.out" using 1:3 title "Worst case" with points, \
     "linearSearch.out" using 1:4 title "Average case" with points
```

The file `linearSearch.pdf` will be saved in the current directory with the contents of the graphics window. To save into a PostScript or JPEG file, you must change the command `set term` with the corresponding parameters. See the *gnuplot* manual.

Notice that the difference between `linearSearch1.plot` and `linearSearch2.plot` are just two commands: `set term` and `set output`. You can run a *gnuplot* command file from the command line as follows:

```
gnuplot linearSearch2.plot
```

Activity 4: approximating functions to empirical results

The `fit` command can be used to obtain approximated functions that show a simpler behaviour of the algorithm. For example, since the worst and average case of linear search present a linear cost, we want to know what is the empirical difference between them, i.e. the associated constants that are not used in the asymptotic notation. Performing the following commands in the *gnuplot* command line we obtain the estimation for the constants:

example of output when running the fit command for average and worst cases

```
f(x)=a*x+b
fit f(x) "linearSearch.out" using 1:3 via a,b
...
Final set of parameters            Asymptotic Standard Error
=======================            ==========================

a               = 0.000904322      +/- 3.84e-05     (4.247%)
b               = -94.818          +/- 23.83        (25.13%)

 ---
g(x)=c*x+d
fit g(x) "linearSearch.out" using 1:4 via c,d
...
Final set of parameters            Asymptotic Standard Error
=======================            ==========================

c               = 0.000403044      +/- 2.468e-05    (6.124%)
d               = -44.592          +/- 15.31        (34.34%)
```

Therefore, we can see that the relation of growth rates between worst case and average case is the relation between the slopes of the corresponding linear functions, i.e., $a/c \approx 2.2$. Consequently, the worst case is twice slower than the average case.

These approximations can also be used for prediction purposes. For example, in the average case, for an array of size 10^9, the time that will be required for the execution is $10^9 \cdot c + d$ milliseconds.

The functions inferred thanks to the *gnuplot* `fit` command can be represented in such a way we can check if the computed values for constants are good approaches or we are wrong. For instance, if the growth rate of a function is approximated by means of a linear function when it should be used a quadratic function.

In order to check the *gnuplot* `fit` command you can run the following command:

```
$ gnuplot linearSearchFit.plot
```

where the contents of file `linearSearchFit.plot` is

linearSearchFit.plot

```
set xrange [0:110000]
set yrange [-20:]
set xtics 200000
set ytics 100
set xlabel "Size"
set ylabel "Microseconds"
set key left
set grid

#set term pdf colour enhanced solid
#set output "linearSearch.pdf"

f(x) = a*x+b
g(x) = c*x+d
```

```
fit f(x) "linearSearch.out" using 1:3 via a,b
fit g(x) "linearSearch.out" using 1:4 via c,d

plot "linearSearch.out" using 1:2 title "Best case" with points, \
     "linearSearch.out" using 1:3 title "Worst case" with points, \
     "linearSearch.out" using 1:4 title "Average case" with points, \
     f(x) with lines, g(x) with lines


print "f(", 10**9, ") = ", f(10**9)
print "g(", 10**9, ") = ", g(10**9)
```

You can uncomment the lines `#set term` and `#set output` in order to save the output into a PDF file.

4 Empirical analysis of sorting algorithms

Students must repeat the empirical analysis for some of the sorting algorithms that were presented in theory lectures, specifically: **insertion sort**, **selection sort**, and **merge sort**.

4.1 Parameters of the empirical study

The parameters to take into account when defining the empirical analysis are the size of the problem, the significant instances and the number of repetitions to be performed in order to obtain significant results. The size of the problem is in all cases the size of the array, and a possible interval is between 1000 and 10000 elements, with increments of 1000. The number of repetitions should be moderate to make the experiment affordable in a lab session; thus, about 200 repetitions is a good number.

4.2 Analysis of Selection Sort

The Selection Sort algorithm sorts arrays performing $\Theta(n^2)$ comparisons where n is the size of the array. This algorithm is no sensible to different instances of the problem for the same input size. So it is only necessary to do the analysis by using randomly filled out arrays.

Activity 5: generating arrays with random values

Add to package `pract3` in your *BlueJ* project `pract3` the provided class `MeasuringSortingAlgorithms.java`, available in *PoliformaT*.
Then, write the code for the method

`fillArrayRandom( int [] )`

before programming the method

`measuringSelectionSort()`

for analysing the behaviour of Selection Sort algorithm. Remember that the code of sorting algorithms is available in the file `MeasurableAlgorithms.java`

```
/* Fills an int array with random values
 * @param a int[], array to be filled
 */
private static void fillArrayRandom( int [] a )
```

Activity 6: running time of a single call to the method

Complete the method `measuringSelectionSort()` in the class `MeasurableAlgorithms` with the necessary instructions to:

1. Create an array `a` of `int` of size 100 by using the method `createArray( size )`.
2. Fill the array `a` by using the method `fillArrayRandom( int [] a )`.
3. Call the method `System.nanoTime()` for getting the current timestamp of the system and store it in a variable `ti` of type `long`.
4. Call the method `MeasurableAlgorithms.selectionSort( int [] )` for sorting array `a`.
5. Call again the method `System.nanoTime()` for getting the current timestamp of the system and store it in a variable `tf` of type `long`.
6. Compute the elapsed time `(tf - ti)` used by the method for sorting the array.
7. Print on screen a line with the size and the elapsed time in microseconds.

Activity 7: running time for a given input size

Taking a unique measure for estimating the empirical temporal cost of an algorithm is not enough. We need to repeat the measure several times and take the average as the empirical temporal cost of an algorithm for a given input size.

It will be better if you define the constant `REPETITIONS` with a value of 200 for instance, in the class `MeasuringSortingAlgorithms` and complete the method `measuringSelectionSort()` with the instructions you wrote in the previous activity.

For each repetition of the measuring process it is recommended to fill again randomly the array. **If you do not do it, all the repetitions but the first one will use a sorted array**. Compute the average of the running times of all repetitions and show it on screen.

Activity 8: running time for different input sizes

At this point you have to define the following constants: `MIN_SIZE`, `MAX_SIZE` and `STEP_OF_SIZE`, with values 1000, 10000 and 1000 respectively.

The method should print to standard output something similar to:

```
# SelectionSort. Time in microseconds
#   Size      Average
#---------------------
    1000       403.346
    2000      1348.255
    3000      3061.948
  ...
```

Activity 9: graphical representation of the results

1. Run the program class `MeasuringSortingAlgorithms`, selecting the Selection Sort option, save the output in a file. Similarly to the case of Linear Search, use *gnuplot* and plot the results by using input sizes for X axis and time in microseconds for Y axis.
2. Make the adjustment of the obtained results by using a quadratic function $f(x) = a * x * x + b * x + c$ in the `fit` command.
3. Plot again the results, but this time including the approximated function. Use the appropriate labels for axes, the plotted values or functions, add a title for the figure and save it into a PDF/JPEG file.
4. Use the approximated function to predict the running time of the algorithm for an array of 800000 integer values.

4.3 Analysis of Insertion Sort

From the theoretical analysis we know the temporal cost function of the Insertion Sort algorithm is bounded by two typical functions. It is lower bounded by a linear function, $T_{IS}(n) \in \Omega(n)$, and upper bounded by a quadratic function, $T_{IS}(n) \in O(n^2)$.

So in the case of this algorithm we have to perform the empirical analysis for the three possible scenarios, best case, worst case and average case.

Activity 10: creating sorted arrays

For analysing the Insertion Sort algorithm we need two auxiliary functions, one to fill arrays in ascending order and another to fill arrays in descending order. First type of arrays to be used in the best case and the others in the worst case. So in this activity you have to complete these methods whose headers are the following:

```
/* Fills an int array sorted in ascending order
 * @param a int[], array to be filled
 */
private static void fillArraySortedInAscendingOrder( int [] a )

/* Fills an int array sorted in descending order
 * @param a int[], array to be filled
 */
private static void fillArraySortedInDescendingOrder( int [] a )
```

Activity 11: empirical analysis of insertion sort algorithm

Complete the method `measuringInsertionSort()` in the class `MeasuringSortingAlgorithms` for performing the empirical analysis of the insertion sort algorithm whose code is available in the method `insertionSort()` of the class `MeasurableAlgorithms`.

The analysis to be performed in this activity should include the running time for best case, worst case and average case for each possible value of the input size. Use the constants defined for evaluating the selection sort algorithm.

The method should show on screen a table similar to the following one:

```
# Insertion Sort. Running time in microseconds
#   Size    Best      Worst    Average
# -------------------------------------
    1000   0.025    422.532    134.647
    2000   0.029    848.167    405.849
    3000   0.040   1904.827    919.622
    ...
```

Notice that for analysing this algorithm in the average case you should fill a new random array at every repetition. Also in the worst case you have to fill an array sorted in descending order at every repetition. If you do not do it results in average and worst case will not be valid. In the case of best case it is not necessary, you do not need to refill the array.

Activity 12: graphical representation of results

- Run the program class `MeasuringSortingAlgorithms`, selecting the Insertion Sort option, and save the output into a file; then, by using *gnuplot*, show the results in a plot where X axis is the input size and Y axis running times. They will represent the points for the best, worst and average cases.

- Make the adjustment by defining functions in all cases. For the best case by means of a linear function, and using a quadratic function for worst and average cases.

  See the obtained values for the coefficients of the three approximating functions.

- Plot again the results with the approximated functions for the three cases. Save it into a PDF or JPEG file.

- Use the approximated functions to predict the running time when sorting an array of 800000 integer values in the worst case (array sorted in descending order), average case (randomly filled out array) and in the best case (array sorted in ascending order).

5 Additional activity: empirical analysis of the Merge Sort algorithm

Complete the method `measuringMergeSort()` in the class `MeasuringSortingAlgorithms` for studying the growth rate of the temporal cost function of this algorithm.

In this case it is suggested to use different input sizes. Redefine the constants `MIN_SIZE` and `MAX_SIZE` for this algorithm and instead of increasing the input size by adding the value of constant `STEP_OF_SIZE`, multiply it by 2 (`size *= 2`) at every iteration of the loop for checking different values of the size.

The maximum size can be 2^{19}. The number of repetitions can be the same used when measuring the other algorithms.

6 Evaluation

This lab activity belongs to the first part of lab practises of this subject, PRG. The first part will be evaluated during the first partial exam. The weight of this part is the 50% of the final lab grade (NPL). Remind that the final lab grade (NPL) is the 25% of the PRG final grade.