Umeå University
Institution för Datavetenskap

Datavetenskapens byggstenar 7.5 p
DV160HT15

OU4 Analysis of Complexity

Submitted	2015-12-25
Author:	Lorenz Gerber (`dv15lgr@cs.umu.se`)
Instructor:	Lena Kallin Westin / Johan Eliasson

Contents

1 Introduction

The aim with this laboration was to apply experimental and asymptotic complexity analysis of algorithms. This type of analysis is used to find out how a algorithm of interest behaves when scaling up the input size. The analysis will answer the question: In which relation is the algorithm runtime to the size of input data. A practical example is to describe the needed time for sorting a list, dependent on the number of list elements. Such an analysis results in a algebraic expression, a function $f(n) = t$ where n is the input size and t the algorithm runtime. This will allow to classify algorithms into a number of broad classes according to the most dominant term in the function: $k, log(n)$, n, n^2, n^k or a^n.

In this laboration, we were experimenting with two common ways for determining the function relation between input size and runtim: experimental complexity analysis and asymptotic complexity analysis based on calculation of primitive operations. While both of those methods can yield rather complex expressions, the aim is to find a simple formula that describes the relation 'on the safe side', i.e. a conservative estimation of run-time. This last description, finding a simple, conservative expression for the relation of input size and runtime qualifies as a 'sloppy' definition of the 'Big O' notation which will be described in more detail further down.

1.1 Experimental Complexity Analysis

In experimental complexity analysis, the time consumption of the algorithm is measured for a number of different input sizes. The value pairs are then visualized in a scatter plot. It is important to choose apropriate ranges of input size n but also a reasonable number of replicate measurments for each input size. The replication is important for mainly two reasons: First, the performance of a computer is not constant. It varies within a small range and measurments will therefore spread somewhat around one theoretical 'true' value. Second, and in most cases more important, many algorithms have a certain 'random' component built in. Alternatively, the test data could be generated randomly to assess for an average case. Taking up the example of sorting a list, the data could be already sorted in the best case or oposite to the requested sort sequence in the worst case. In any case, replicate measurments for the same input size are inavoidable.

After plotting the experimental data, assumptions about the relation between input size and runtime are made. They can be visually tested by transforming the experimental data with the inverse of the expected function. If, for example a graph looks like a second degree polynom, transforming the response variable with the square-root should yield a straight line, which visually is much easier to identify as such than a curve. If the assumption was correct and the resulting graph looks linear, a linear regression analysis will numerically provide proof for the assumption. The obtained line equation can then be transformed back, in the described case resulting in a second degree polynom. From here, the next step will be to determine 'ordo' of the obtained function which will be described further down.

1.2 Asymptotic Complexity Analysis

In asymptotic complexity analysis, the aim is the same as described above in experimental complexity analysis: Obtaining a algebraic relation between input data size and runtime. However, in asymptotic complexity analysis, the algorithm must be know and available. The analysis is based on a detailed account for how often each step in the algorithm will run as a function of input size n. There can be certain steps in an algorithm, that will be independent of input size, constant. Others will be related linear, polynomal, logarithmic or exponential with input size. A Typical example is a double `For` loop: It will yield a $n/timesn$ relation, hence it results in a n^2 funtion.

The steps to account for in an algorithm are called 'primtive operations'. This is a rather theoretical definition that leaves some room for interpretation. But in general, primitive operations are `[read]`, `[write/assign]`, `[compare]`, `[increase/decrease]`, `[arithmetic operations]` etc. Summing up all primitive operations for a given algorithm will result in a function $f(n)$ where n is the input size.

1.3 The 'Big O' notation'

'Ordo' or 'Big O' notation is a mathematical definition of an algorithms time complexity. It's formal defintion is shown in equation (1)[1, pp. 245].

$$f(n) \Rightarrow O(g(n)) \text{ if } f(n) \leq c \times g(n) \text{ for } n \geq n_0 \text{ and } c > 0 \text{ and } n_0 \geq 1 \tag{1}$$

In words: If the function $c \times g(n)$ for n larger than n_0 is always bigger than $f(n)$, it is $O(g(n))$ (say: *ordo g* of n). Hence knowing 'ordo' of an algorithm gives a good estimation of it's time complexity.

1.4 Determining of *Ordo*, c and n_0

Determing 'Ordo' is done in several steps. First the most dominant function terms of $f(n)$ has to be found and set as $g(n)$. In a general second degree polynom, n^2 is for example the dominant term.

Then the constant factor c is determined by calculating the limes from n to infinity of dividing $f(n)$ by $g(n)$ according to equation (2).

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} + 1 \tag{2}$$

The value n_0 is the lower limit of input size n for which $g(n)$ will be at least equal or larger than $f(n)$. It is found by solving equation 3.

$$c \times g(n) = f(n) \tag{3}$$

2 Material and Methods

2.1 Experimental Complexity Analysis

We obtained a compiled program to run from the command line. The program would take three arguments: a selector for one of two implemented algorithms, size of in data and number of repetitions. It was unknown to us which algorithms were implemented in the program. The program would run a certain time and before quitting, print the runtime to standard out.

To automate the experimental process, a shell script (listing 1) was written that run the program 10 times for the input sizes n = (1'000, 2'500, 5'000, 10'000, 15'000, 20'000, 25'000, 50'000, 75'000, 100'000, 125'000, 150'000, 200'000).

The script was run from the bash commandline on a MacbookPro8,1 and the output piped and appended to a text file which was then imported to statistical computing environment R [3]. All data analysis and plotting was done in R.

Listing 1 Shell script to automate the collection of experimental data. Standard output was piped and appended to a text file.

```
#!/usr/local/bin/bash
ARRAY=(1000 2500 5000 10000 15000 20000 25000 50000 75000  100000 125000 150000 200000)
for a in ${ARRAY[*]}
do
    for b in {1..10}
    do
        ./algorithms_osx -2 $a
    done
done
```

2.2 Asymptotic Complexity Analysis

We were given pseudo code of a bubble sort algorithm (listing 1) to conduct asymptotic complexity analysis. A short description on how to account for primitive operations was available through the course homepage [2].

Listing 2 The given pseudo code of a bubble sort.

```
Algorithm bubbleSort(numElements, list[])
input:  numElements, the number of elements in the list
        list, a list of numbers to be sorted
output: the sorted list

    done <- false
    n <- 0
     while (n < numElements) and (done = false)
        done <- true
        for m <- (numElements -1) downto n
            if list[m] < list[m - 1] then
                tmp <- list[m]
                list[m] <- list[m - 1]
                list[m - 1] <- tmp
                done <- false
       n <- n + 1
    return list
```

After accounting for primitive operations and obtaining algebraic functions describing ‘best’ and ‘worst’ case for the bubblesort algorithm, ‘ordo’ expressions were determined according to lecture notes.

3 Results

3.1 Experimental Complexity Analysis

The left panel of figure 1 shows the experimental runtime values plotted against *n* length of input data. The values were then transformed by calculating the square root. The trans-

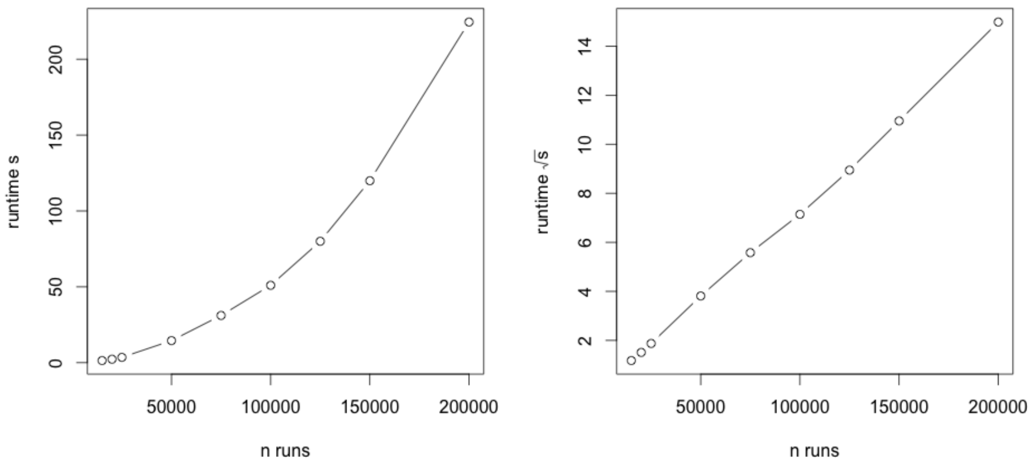

Figure 1: *Runtime values in seconds for a series of n repetitions of the investigated algorithm. The left panel shows the measured times. For the right panel, the square root for each time value was calculated before plotting.*

Table 1 *Linear regression of the transformed experimental data*

gradient	7.348×10^{-5}
intercept	1.393×10^{-2}
R^2	0.9988

formed data is shown in the right panel of figure 1. Then linear regression was calculated on the transformed data (*table1*).

The obtained linear equation was transformed back to yield a quadratic function as shown in equation (4).

$$y = 5.4 \times 10^{-9} x^2 + 2.1 \times 10^{-6} x + 0.2 \times 10^{-4} \tag{4}$$

Then c was determined according to equation (2) and n_0 was determined by solving the quadratic equation resulting from $f(n) = g(n)$ (Table 3, figure 2).

3.2 Asymptotic Complexity Analysis

Worst Case

Listing 3 shows the determination of primitive operation for the ‘worst’ case. The line numbers correspond to the line numbers in the original pseudocode shown in listing 1.

Table 2 *Calculated value for ordo determination*

c	6.4×10^{-9}
n_0	2060

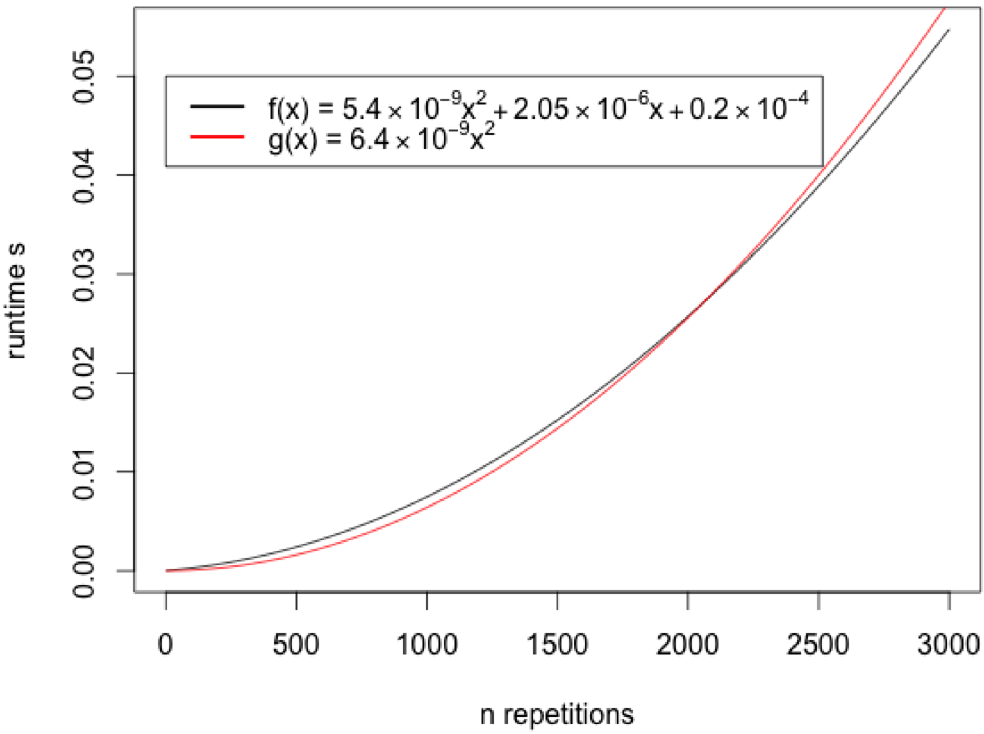

Figure 2: *Comparison of $f(x)$ and $O(g(x))$ in the range of n_0 which was determined to be at about $n = 2600$.*

Listing 3 *Determining the 'worst case' complexity for the given 'bubblesort' algorithm. The line numbers correspond to those in listing 1.*

```
1:  1 * [<-] +
2:  1 * [<-] +
3:  (numElements + 1) *
    (3 * [get] + 1 * [<] +  1 * [=] + 1 * [AND]) +
4:  numElements * (1 * [<-]) +
5:  init:
    (numElements * (numElements -1) / 2 + 1) *
    (1 * [get] + 1 * [-] + 1 * [<-]) +

    cond success + counter:
    numElements * (numElements - 1) / 2) *
    (2 * [get] + 1 * [>] + 1 * [--]) +

    cond fail:
    numElements *
    (2 * [get n] + 1 * [>]) +

6:  (numElements * (numElements - 1) / 2) *
    (2 * [get] + 2 * [list[]] + 1 * [-] + 1 * [<] +
7:      1 * [get] + 1 * [list[]] + 1 * [<-] +
8:      1 * [get] + 1 * [-] + 1 * [list[]] + 1 * [<-] +
9:      2 * [get] + 1 * [-] + 1 * [ <-] +
10:     1 * [<-] ) +
11: numElements * (1 * [get] + 1 * [+] + 1 * [<-] +
12: 1 * return

set numElements = x

1:  1 +
2:  1 +
3:  (x + 1) * 6
4:  x * 1
5:  (x * (x-1) / 2 + 1) * 3 +
    (x * (x - 1) / 2) * 4 +
    x * 3 +
6:  (x * (x - 1) / 2) * (6 +
7:    3 +
8:    4 +
9:    4 +
10:   1) +
11: x * 3 +
12: 1

Hence:
1 + 1 + 6x + 6 + x + 1.5x^2 - 1.5x + 3 + 2x^2 - 2x +
3x + 9x^2 - 9x + 3x + 1

= 12.5x^2 + 4.5x + 12
```

Best case

Listing 4 shows the determination of primitive operation for the 'best' case. The line numbers correspond to the line numbers in the original pseudocode shown in listing 1.

Listing 4 Determining the 'best case' complexity for the given 'bubblesort' algorithm. The line numbers correspond to those in listing 1.

```
1:  1 * [<-] +
2:  1 * [<-] +
3:  2 * (3 * [get] + 1 *[<] + 1 * [and] + 1* [==]) +
4:  1 * [<-] +
5:  init:
    1 * [get] + 1 * [-] + 1 * [<-] +

    cond success + counter:
    (numElements - 1) * (1 * [get] + 1 * [>]) + 1 * [--]) +

    cond fail:
    1* [get] + 1 * [>] +
6:  (numElements - 1) *
    (2 * [get] +  2 * list[] + 1 * [-] + 1 * [<]) +
11: 1 * [get] + 1 * [+] + 1 * [<-]
12: 1 * [return]

set numElements = x

1:  1 +
2:  1 +
3:  2 * 6 +
4:  1 +
5:  3 +
    (x - 1) * 3 +
    2 +
6:  (x - 1) * 6 +
11: 3 +
12: 1

Hence:
1 + 1 + 12 + 1 + 3 + 3x - 3 + 2 + 6x -6 + 3 + 1
= 9x + 15
```

In table 3 the keyfigures of the 'ordo' determination for 'best' and 'worst' case expressions can be found.

Table 3 *Determination of ordo for 'Best' and 'Worst Case'*

	Worst Case	Best Case
$f(n)$	$12.5n^2 + 4.5x + 12$	$9x + 15$
$g(n)$	$13.5n^2$	$10x$
c	13.5	10
n_0	7	15

4 Discussion

4.1 Experimental Complexity Analysis

4.2 Asymptotic Complexity Analysis

Worst Case

Line 1, 2 and 12 run just once. Line 3 runs `numElements + 1` times. Lines 4 and 11 run `numElements` times. Line 5 runs `(numElements * (numElements - 1)) / 2) + 1` times. The lines 6 to 10 run `numElements * (numElements - 1) / 2` times. The `downto` in `for m <- (numElements - 1) downto n` was interpreted as 'larger than' condition.

References

[1] L.E. Janlert and T. Wiberg. *Datatyper och algoritmer*. Studentlitteratur, 2000.

[2] N. Lockner. Komplexitetsanalys - exempel. Umeå University, Institute for Computer Science, 2 2012.

[3] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2015.