Algorithms & Data Structures

Dr Michele Sevegnani

Joao Almeida-Domingues[*]

University of Glasgow

January 14th, 2020 – March 25th, 2020

Contents

These lecture notes were collated by me from a mixture of sources , the two main sources being the lecture notes provided by the lecturer and the content presented in-lecture. All other referenced material (if used) can be found in the *Bibliography* and *References* sections.

The primary goal of these notes is to function as a succinct but comprehensive revision aid, hence if you came by them via a search engine , please note that they're not intended to be a reflection of the quality of the materials referenced or the content lectured.

Lastly, with regards to formatting, the pdf doc was typeset in LaTeX, using a modified version of Stefano Maggiolo's class

[*]2334590D@student.gla.ac.uk

1 Analysis Techniques

running time , experimental analysis , theoretical analysis , primitive operations , growth rate of running time , Big-Oh

1.1 definition. Data Structure is a systematic way of organising and accessing data

1.2 definition. Algorithm is a step-by-step procedure for performing some task in a finite amount of data

1.3 definition. Running Time is the amount of time it takes an algorithm to execute in full

The main criteria used to compare different algorithms are running time and memory usage. In general the running time of an algorithm increases with the input size, and may vary depending on the hardware and software environments on which it is run. When comparing algorithms one aims to control those variables and express a relation between their run times and inputs via some function

1.1 *Experimental Analysis vs Theoretical Analysis*

One way to study the efficiency of an algorithm is to run an empirical study. The dependent and independent variables are set, several trials are run a with appropriate inputs and then a statistical analysis is carried out on the output of the experiments.

Drawbacks

Though the experiment itself is relatively straightforward to run (once the algorithm is implemented), the analysis can be quite quite complicated to perform. Especially given that it can be hard to know which inputs are appropriate to use and , more importantly , given that fully implementing a complex algorithm is hard work and time-consuming. Hence, if possible a higher-level analysis is performed, if an algorithm can be deemed to be inferior by application of theoretical methods then no experimental analysis is required

So, when developing theoretical methods of analysis one's goal is to overcome the drawbacks mentioned above in order to achieve the following:

1. System Independence
2. Input Coverage
3. High-Level Description

1.4 definition. Primitive Operations are *low-level* instructions with constant execution time (e.g. variable assignment, function call)

In order to express running time as a function of input size we use primitive operations, which are identifiable from abstract implementations (like pseudocode) and taken to have a constant time of completion. In this way, one

can compare the total number t of ops for a given implementation, since by assuming a constant time for all ops one can assume that the total running time will be *proportional* to t

Ideally the average of all possible inputs would be used to characterize a given algorithm. However,finding the average often involves finding an appropriate distribution for the sample inputs. Hence, we characterize it instead in terms of its *worst case* input which is far easier to identify.

1.5 remark. Another advantage is that minimising for the worst case by definition implies that we're optimising the running time for all other possible inputs

1.2 *Common Functions*

This relation between input and running time is often expressed in terms of one of the following 7 functions

Constant

The simplest function of them all is the constant function which simply assigns a given constant c to any input n . It is particularly useful, since it allows us to express the *number of steps* needed to perform a basic operation

$$f(n) = c, \forall n \in \text{ input set}$$

1.6 remark. The essential constant functions if $g(n) = 1$ given that we can express any other constant function in the form $g(n)f(n)$

Logarithm

The logarithmic function, in particular $\log_2$, pops up all the time. A logarithmic runtime is characterized by larger differences in runtime for smaller inputs, with a significant decrease for larger inputs.

1.7 definition. ceiling (of x) the smallest integer greater than x

1.8 notation. $\lceil x \rceil$

We can think of the ceiling function as an approximation of x. We can use it in a similar manner to approximate any given algorithm. By definition $\log_b(x)$ is just the power to which b has to be raised to give x. Hence, we define $\lceil \log_b x \rceil$ as the smallest number for which b has to be raised so that it includes x. So, we can divide repeatedly divide x by b until we get a number less or equal to 1

1.9 example. $\lceil \log_2 12 \rceil = 4$ since $2^3 < 12 < 2^4$

Linear

The linear function assigns the input to itself. It is useful to characterize single basic operations on n elements

$$f(n) = n$$

N-Log-N

For any given input n it assigns n times $\log(n)$

$$f(n) = n\log(n)$$

1.3 *Quadratic*

The quadratic function appears primarily in algorithms with nested loops, since the inner loop performs an operation n times and the outer loop will repeat each loop a *linear* number of times

$$f(n) = n^2$$

Polynomials

A more general class which subsumes the quadratic ($d = 2$), linear ($d = 1$) and constant ($d = 0$) functions, where d is the degree of the polynomial which corresponds to the largest exponent in the polynomial expression. In general, polynomials with lower degrees have better running times

$$f(n) = \sum_{i=0}^{d} a_i n^i$$

Exponential

Exponential time polynomials are generally the worst-case running time for an algorithm, since they grow very rapidly. As an example, take a loop which doubles the number of operations it performs with every iteration. Then, the total number of operations it performs will be 2^n

1.4 *Growth Rates*

If we take $f(n)$ to be the function which gives the total number of operations in the *worst-case*, and a z to be the time taken by the fastest and slowest primitive op , respectively. Then, the worst-case running time $T(n)$ for the algorithm is bounded by the two linear functions $af(n); zf(n)$, i.e

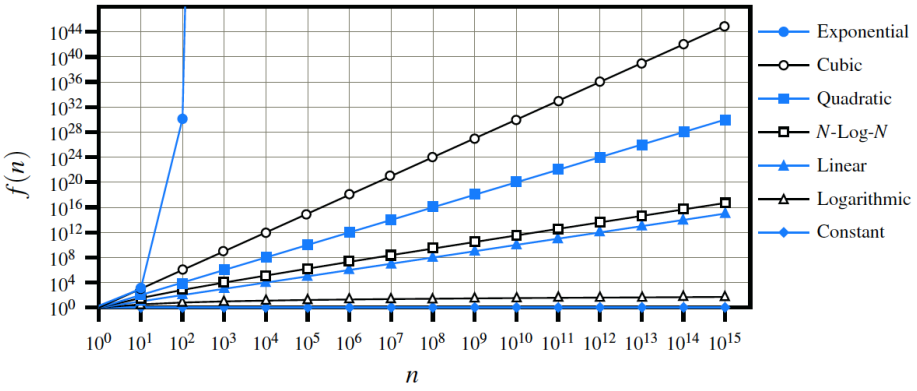
$$af(n) \leq T(n) \leq bf(n)$$

Note that the growth rate of the worst-case running time is an intrinsic property of the algorithm which is not affected by by hardware or software environments. These factors affect $T(n)$ by a constant factor. Hence, when

running an asymptotic analysis one can **disregard** constant and lower-degree terms

1.10 remark. Ideally we want operations on data structures to run in times proportional to the constant or log functions, and algorithms to run in linear or n-log-n

1.11 definition. Asymptotic Analysis analysing how the running time of an algorithm increases with the size of the input *in the limit*, as the size of the input increases with the bound



1.5 *Big-Oh Notation*

1.12 definition. Big-Oh Notation we say that *"$f(n)$ is big-Oh of $g(n)$"* if $\exists c \in \mathbb{R} s.t f(n) \leq cg(n)$, for $n \geq n_0$

1.13 notation. $f(n) isO\,(g(n))$

In essence we make use of the fact that growth run time is not affected by constant factors, and encode this into function notation. In effect bounding the function after a given input size n_0 by another function ; i.e $f(x)$ is strictly less than or equal to $g(x)$ up to to a constant factor and in the *asymptotic sense*. Hence, we can use $g(x)$ to approximate/characterize $f(x)$

We want this relation to be expressed in the simplest terms possible. Obviously it is easy to find such a function if one aims for the moon, i.e clearly $3n^2 + 2$ is $O(n^{10})$ this is however not very useful. Instead, simplifying the first expression by getting read of constant factors , we see that we can be sure that it is for sure $O(n^2) < O(n^{10})$, which is therefore a better approximation

1.14 remark. We want the *simplest* expression of the class of bounding functions (n^2 Vs $4n^2$) and the *smallest* possible class (n^2 Vs n^{10})

1.15 proposition. *For any polynomial $p(n)$ of degree d , $p(n) \in O\left(n^d\right)$*

1.16 proposition. $T_1(n) \in O(f(n)), T_2(n) \in O(g(n)) \implies T_1(n) + T_2(n) \in O(max(f(n), g(n)))$

1.17 proposition. $T_1(n) \in O(f(n))$ *and* $T_2(n) \in O(g(n)) \implies T_1(n)T_2(n) = O(f(n)g(n))$

Proof. $T_1T_2 = klf(n)g(n)$, for constants k, l . Hence, ignoring the constants we have the expected result

1.18 proposition. $T(n) = (\log n)^k \implies T(n) = O(n)$

Proof.

$$T(n) \in O(f(n)) \iff \lim_{n\to\infty}(T(n)/f(n)) = 0)$$

By *L'Hopital's*,

$$lim_{n\to\infty}(f(n)/g(n)) = lim_{n\to\infty}(f\prime(n)/g\prime(n))$$

Take $f(n) = (\log n)^{k+1}$ and $g(n) = n$. Then, by induction on k ,

$$(\log n)^1 = \log n \in O(n) \text{ and } (\log n)^k \in O(n)$$

Hence,

$$\lim_{n\to\infty}\left(\frac{(\log n)^{k+1}}{n}\right) = 0$$

$$\lim_{n\to\infty}\left(\frac{(\log n)^{k+1}}{n}\right) = \lim_{n\to\infty}\left(\frac{(k+1)(\log n)^k}{n}\right) = 0$$

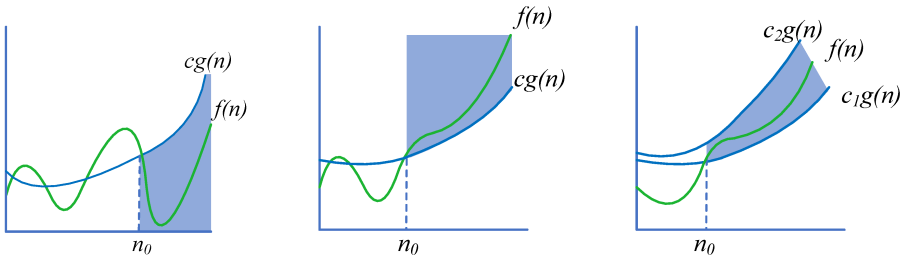
Therefore, the result follows by induction

1.6 *Big-Omega & Big-Theta*

1.19 definition. $\Omega(n)\ f(n) \geq cg(n)$

1.20 definition. $\Theta(n)\ c\prime g(n) \leq f(n) \leq c\prime\prime g(n)$

Similarly to $O(n)$, these provide upper and upper+lower bounds respectively



(a) Big-O Notation (b) Big-Omega Notation (c) Big-Theta Notation

Figure 1: Asymptotic Notation `shorturl.at/xAFQ1`

1.7 *Computing Running Times*

The following, follows from the propositions above

1. **Loops :** at most the running time of its contents times the number of iterations

2. **Nested Loops :** it should be analysed inside out. Take the runtime of the expression and multiply it by the product of of the sizes of all the loops

3. **Consecutive Statements :** add

4. **If-then-else :** at most the time of the test condition with the maximum of the running times of the two branches

1.21 example.

```
sum = 0;
for (i=1; i<=n; i++)
        sum += n;
```

See more examples here

Analysis of Insertion-Sort

1.22 example. Insertion-Sort

Algorithm 1: Insertion-Sort

Data: Array of integers : A
Result: Permutation of A such that A[0] $\leq$ A[1] $\leq \ldots$ A[n-1]

```
1  Insert(A)
2  for j = 1 to n − 1 do                                   /* n */
3      key := A[j]                                         /* n − 1 */
4
5      i := j-1                                            /* n − 1 */
6
7      while i ≥ 0 and A[i] > key do                       /* Σ_1^{n−1} t_j */
8          A[i+1] := A[i]                                  /* Σ_1^{n−1}(t_j − 1) */
9
10         i := i-1                                        /* Σ_1^{n−1}(t_j − 1) */
11
12     end
13     A[i+1] := key                                       /* n − 1 */
14
15 end
```

Hence, summing the individual operations

$$T(n) = n + (n-1) + (n-1) + \sum_{1}^{n-1} t_j + \sum_{1}^{n-1} (t_j - 1) + \sum_{1}^{n-1} (t_j - 1) + (n-1)$$

Best Case: A is already sorted , which means that the while loop is never executed and $t_j = 1$. Hence,

$$T(n) = n + (n-1) + (n-1) + (n-1) + 0 + 0 + (n-1) = O(n)$$

Worst Case: At every iteration we shift j elements : $t_j = j$. So,

$$\sum_{1}^{n-1} t_j = \sum_{1}^{n-1} j = \frac{n(n-1)}{2} = O(n^2)$$

$$\sum_{1}^{n-1} t_j - 1 = \sum_{1}^{n-1} (j-1) = \frac{n(n-1)}{2} - (n-1) = O(n^2)$$

Hence,

$$T(n) = n + (n-1) + (n-1) + O(n^2) + O(n^2) + O(n^2) + (n-1) = O(^2)$$

References

CS2 Software Design & Data Structures
calculating˙program˙running˙time

CS2 Software Design & Data Structures. URL: https://opendsa-server.cs.vt.edu/ODSA/Books/CS2/html/AnalProgram.html.

Goodrich et al.: Data Structures and Algorithms in Java, 6th Edition
goodrich˙tamassia˙2014

Michael Goodrich and Roberto Tamassia. *Data Structures and Algorithms in Java, 6th Edition*. John Wiley & Sons, 2014.