COMP20230: Data Structures & Algorithms **2018-19 Semester 2**

Lecture 4: Jan 30

Lecturer: Dr. Andrew Hines *Scribes: Avik Saha, Giovanni Facchinetti*

4.1 Outline

This session introduces how the effectiveness and efficiency of an algorithm can be measured/calculated by using time complexity (running time of an algorithm). The time taken by a program to complete its execution depends on a number of factors - i.e. the hardware, the operating system of the machine, the software itself. But finding the time complexity of an algorithm ensures that the emphasis is put only on the algorithm and not on any other external factors. The running time of any algorithm varies with the input and typically grows as the size of the input increases.

4.2 Big-$\mathcal{O}$, Big-Ω, Big-Θ

The analysis of any algorithm can lead us to 3 cases- *worst*, *best* and *average*. In most of our practical analysis the emphasis is on the worst-case scenario, as we would want to know in advance how the algorithm behaves in a worst case situation.

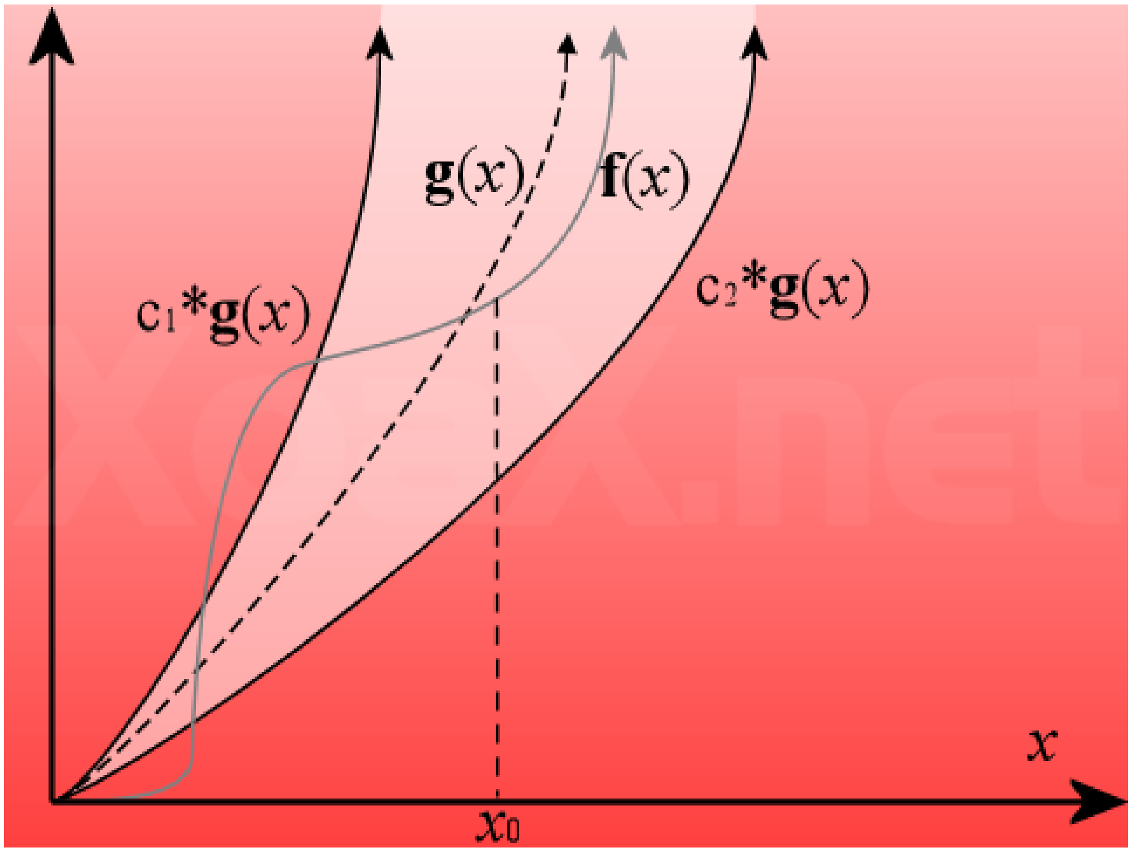
Big-$\mathcal{O}$ measures the worst case time: asymptotic analysis for Big-$\mathcal{O}$ allows us to understand the worst case algorithm speed.
Consider a generic function f(n) where n represent an input size. The Big-$\mathcal{O}$ of f(n) is measured based on a function c*g(n), where c*g(n) is the closest, tightest upper bound of f(n). Mathematically, it can be represented by f(n)$\leq$ c*g(n).

If n' is a point in n for which c is greater than 0, if the condition f(n)$\leq$ c*g(n) is satisfied, then we say time complexity of f(n) is $\mathcal{O}$(g(n)).

Note: c and n' are real numbers where c is greater than 0 and n' is greater than equal to 1.

Big-Θ measures the best case scenario: i.e., time.
Consider a function f(n). The Big-Θ of f(n) is measured by using a function c*g(n) where c*g(n) is the closest, tightest lower bound of the function f(n). Mathematically it can be represented by f(n) $\geq$ c*g(n).

Big-Ω measures the average case time.
Consider a function f(n). We say f(n) is Θ(g(n)) if f(n) is both $\mathcal{O}$(g(n)) and Ω(g(n)) i.e. f(n) grows the same as g(n) (tight-bound).



The figure above shows an example of a function f(x) and:

- its upper bound function c_1*g(x)
- its lower bound function c_2*g(x)
- its average function g(x)

4.3 Time Complexity Analysis

4.3.1 Constant Running Time

Consider a function f(n) = c, where:
n is the input size and
c any arbitrary constant.
Here, irrespective of the size of n, the function f(n) will always yield a constant value that is independent of n.
For example, initializing a variable x = 5(/10/15....any number) implies that only one single operation is conducted. The same result is produced when returning an array index from a function: return array[1] means 2 operations are conducted (one for accessing the array index and the other for returning the same). In both the examples it is evident that n could be 10,100...or a million but it has no impact on the aforementioned operations. The constant function characterises the number of steps to perform a primitive operation (simple arithmetic, assignment, array access etc.) on a computer.

The time complexity for constant running time is thus given by $\mathcal{O}(1)$.

4.3.2 Linear Running Time

Given an input n, the linear function always assigns the value n itself f(n) = n. This represents that n operations need to be performed to read a number n of elements from the memory. This kind of scenario arises typically when we need to iterate over n number of items or perform any operation over n elements (e.g.: for-loop)

Algorithm 1: Return the sum of n elements

```
function argsum(array): Input  : an array of size n
Output: the index of the maximum value
index ← 0 #1 op: assignment
sum ← 0 #1 op: assignment
foreach i in [1, n-1] do #2 ops per loop
sum ← sum+i #2 ops per loop
endfor
return sum #1 op: return
```

The number of operations needed is in linear and direct proportionality to the size of the input(n): in the *Algorithm 1* above, if we sum all the elementary operations needed we obtain a function 4n+3. The time complexity T(n) for 4n+3 is given by $\mathcal{O}(n)$.

4.3.3 Quadratic Running Time

An algorithm is said to be of *polynomial* time if its running time is upper-bounded by a polynomial expression on the size of the algorithm input, i.e.: T(n) = O(n^k) for some positive constant k.
This is common in algorithm analysis, especially in presence of *nested loops*.
An algorithm is said to be quadratic time if the time complexity T(n) = $\mathcal{O}(n^2)$.
E.g.: simple, comparison-based sorting algorithms are quadratic (e.g. insertion sort).

Algorithm 2: Return possible products of the numbers in an array

```
1 function argprod(array): Input  : an array of size n
  Output: list of all possible products between elements in the list
2 products ← [] #1 op: make an empty list
3 sum ← 0 #1 op: assignment
4 for i in [0, n-1] do #2 ops per loop
5 for j in [0, n-1] do #2 ops per loop per loop
6 products.append(array[i] * array[j]) #4 ops per loop per loop
7 endfor
8 endfor
9 return products #1 op: return
```

So the number of operations needed for the above *Algorithm 2* is given by: $6n^2+2n+2$ and the time complexity T(n) for the same is given by $\mathcal{O}(n^2)$.

4.3.4 Logarithmic Running Time

Given a positive constant b>0, we define the logarithmic function f(x)=$\log_b n$ x=$\log_b n$ if and only if b^x=n. Since the exact determination of a *logarithm* involves calculus and may result not straightforward, we choose to approximate it to the smallest integer greater than or equal to $\log_a n$.
We can regard this value as the number of times we can divide n by a until we get a number less than or equal to 1.
Since we are analysing algorithms that run on computers that operate with binary logic, base 2 algorithms are most common in the computer science domain - in fact, base-2 approximation of a log appears a lot in algorithm analysis, since in programs it is common to repeatedly divide an input in a half.
To give an example, we consider the *logarithm* in base 2 of 12: to approximate its value, we proceed by dividing the log argument (12) by the base until we reach a number that is less than 0. $\log_2 12$ is the expression we want to approximate, where 2 is the base and 12 is the argument
$12/2 = 6$
$6/2 = 3$
$3/2 = 1.5$
$1.5/2 = 0.75 \Rightarrow$ we stop here, as $0.75 \leq 1$
We can thus say that $\log_2 12 \simeq 4$
NB: From now on, we will refer to $\log n$ as to $\log_2 n$.
Logarithm function gets slightly slower as n grows. Whenever n doubles, the running time increases by a constant. A common example of a logarithmic running time is the **binary search technique** - a search algorithm that finds the position of a target value within a sorted array.

Looking at *Algorithm 3* below: after the initial call of the *binarysearch* function, we are interested in the **worst case scenario**: the *elem* we are looking for is never found. Counting the elementary operations is not necessary in this case, as after the first step we have divided our search field in two halves ($n/2$); we then proceed and divide each half by 2 until we reach the base case, so that n will be divided by 2^k.
The last iteration will be when: $1 \leq \frac{n}{2^k}$ and $\frac{n}{2^{k+1}} < 1$.

Assuming this as the worst case (we couldn't proceed any further), we can say that **the time complexity of a binary search algorithm is $\mathcal{O}(\log n)$.**

Algorithm 3: Return index of an item in an array

```
function binarysearch(myarray, elem):
Input   : a sorted array, our search element
Output: the index of the element we want to find
low $\leftarrow$ 0 high $\leftarrow$ n-1 while low $\leq$ high do    mid $\leftarrow$ (low + high)/2
  if myarray[mid] greater than elem then
    high $\leftarrow$ mid-1
    else
    if myarray[mid] ¡ elem then
      low $\leftarrow$ mid + 1
      else
      return mid
    endif
  endif
return size of myarray + 1 #to show the elem is not in the array
```

4.4 Big-$\mathcal{O}$ Rules

When analysing an algorithm, we want to ultimately focus on how its running time grows depending on the input size n: we can safely assume that the running time of an algorithm such as a linear search over a n-dimension array grows proportionally to n - the ”true” running time will be n times a constant that depends on other drivers, i.e. the specific machine used for testing purposes.
With Big-$\mathcal{O}$ notation, our aim is to characterize a function as closely as possible in its upper-bound.

To give an example, consider the function:
$f(n) = 5n^3 + 2n^2 + 7$
We might safely say that is $\mathcal{O}(13n^8 + 35n^5 + 3)$ or even $\mathcal{O}(49n^4)$.
Nevertheless, it is way more accurate to say that $f(n)$ is $\mathcal{O}(n^3)$. There is no actual need to include constant factors and/or lower order terms to characterize our function in terms of asymptotic growth. We can thus generalize the above examples in **some general Big-$\mathcal{O}$ rules**:

1. **Forget the lower order terms and constant factors**
   Given a $f(n)$ polynomial expression of degree d: $f(n)$ is $\mathcal{O}(n^d)$
2. **Always consider the smallest/closest possible degree**
   ”$2n$ is $\mathcal{O}(n)$” is more helpful than ”$2n$ is $O(n^2)$”
3. **Use the simplest expression of the class**
   ”$3n + 5$ is $\mathcal{O}(n)$” is preferable than ”$3n$+5 is $\mathcal{O}(3n)$”

Side notes:
Very large constant factors can lead to some misunderstanding. E.g. $9592n$ is $\mathcal{O}(n)$ but if we compare it with the running time $10n \log n$, we would prefer $\mathcal{O}(n \log n)$, even though the linear-time algorithm is faster when stretched to asymptotic levels of our input size n. On the other hand, a $\mathcal{O}(n^2)$ may be consistently fast given a small n, but again we test the algorithm with ideally large inputs so a quadratic curve will always be less efficient than a linear one.

4.5 Estimating Big-$\mathcal{O}$

As a ”rule of thumb” we could summarize the following steps:

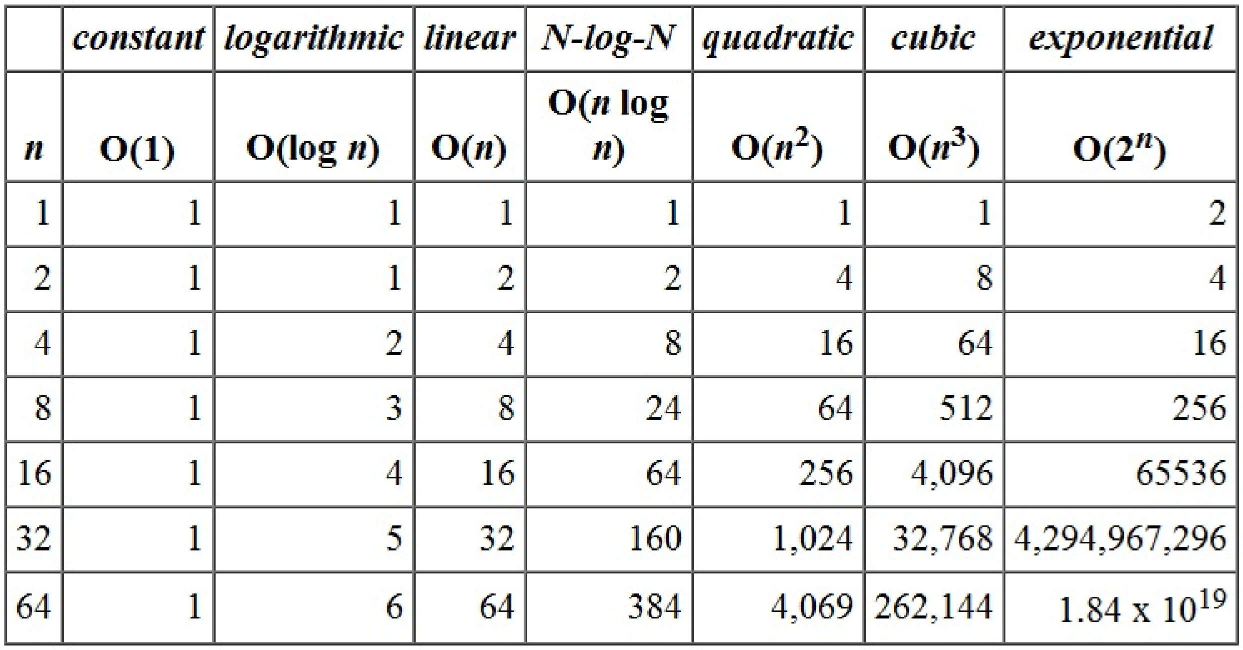
- first of all, check loops and recursion in the algorithm
- if the loop is run over a fixed range [i-n], it will typically be $\mathcal{O}(n)$
- in presence of nested loops, the algorithm will typically be $\mathcal{O}(n^2)$
- when it is not obvious, use induction technique

4.6 Open issues

Big-$\mathcal{O}$ is an useful technique to analyze an algorithm's quality in terms of speed and efficiency.
Apart from speed and efficiency, an algorithm should also be characterized in terms of space complexity (consumption of memory) and power consumption. We can assume that the latter is positively correlated with the number of operations carried out.
But we have not covered (yet) other fundamental attributes of a good algorithm, for which ad-hoc tests are necessary:

- correctness (unit testing)
- security (penetration testing)
- robustness (performance testing)
- clarity (code coverage)
- maintainability (code reviews)

The table below shows the most frequent time complexities occurring in algorithms: it provides numerical examples with increasing n input sizes.
It is worth noting how inefficient are algorithms with exponential time complexities, and the efficiency of binary search algorithms - characterized by $\mathcal{O}(\log n)$.



	constant	*logarithmic*	*linear*	*N-log-N*	*quadratic*	*cubic*	*exponential*
n	**O(1)**	**O(log *n*)**	**O(*n*)**	**O(*n* log *n*)**	**O(n^2)**	**O(n^3)**	**O(2^n)**
1	1	1	1	1	1	1	2
2	1	1	2	2	4	8	4
4	1	2	4	8	16	64	16
8	1	3	8	24	64	512	256
16	1	4	16	64	256	4,096	65536
32	1	5	32	160	1,024	32,768	4,294,967,296
64	1	6	64	384	4,069	262,144	1.84×10^{19}

4.7 References

This lecture notes have been written taking inspiration from:

- lecture slides from Dr. Andrew Hines
- https://www.cpp.edu/f̃tang/courses/CS240/lectures/analysis.htm

The pictures have been taken from:

- https://www.cpp.edu/f̃tang/courses/CS240/lectures/img/alg-tab.jpg
- http://xoax.net/comp_sci/crs/algorithms/lessons/Lesson6/Image1.png