MAA507: Lecture 2 Computational complexity and some Graph Theory

Lars Hellström

January 22, 2014

Contents of todays lecture

An important concern when choosing the algorithm to use for something (after basic requirements such as *correctness* and *stability*) is the amount of various resources that the algorithm will require.

The most important resources are:

- **work** How many primitive operations would it take to carry out the algorithm?
- **time** How long time would it take to execute the algorithm?
- **space** How much memory would the algorithm need to use?
- **energy** How much energy is required to run the algorithm? (Emerging concern!)

In the classical single processor setting, time and work are pretty much the same, so one often speaks of time complexity (or just complexity) when discussing the amount of *work*. Parallel processing is growing in importance, but considering it is still viewed as luxury.

What is asymptotic complexity?

The resource requirements of an algorithm depend on what input data is is given.
Predicting the exact resource requirements is seldom feasible, but it is often possible to give useful bounds for them.
The main parameter in terms of which such bounds are stated is the size of the input, i.e., the number of bits that would be required to encode the input.
If n is the input size and $f(n)$ is some bound on a resource, then the asymptotic complexity of that resource is the asymptotic behaviour of the function $f(n)$ as $n \to \infty$.

Big O notation

The main workhorse in describing asymptotic behaviour is the big O notation:

$$f(n) = O\big(g(n)\big) \qquad \text{(read '}f(n) \text{ is O of } g(n)\text{')}$$

means

> *there exists positive constants M and N such that $\big|f(n)\big| \leqslant M\big|g(n)\big|$ for all $n > N$.*

(Note the abuse of $=$, which should really be an $\in$.)
If $g(n)$ is nonzero for sufficiently large n, the above is equivalent to

$$\limsup_{n\to\infty} \left|\frac{f(n)}{g(n)}\right| < \infty.$$

Big O is a way of ignoring constant factors (which anyway depend on what units one uses):

$$Cg(n) = O\big(g(n)\big) \qquad \text{for all } g.$$

Big O also lets one ignore lower order terms:

$$f(n) = O\big(g(n)\big) \quad \Longrightarrow \quad g(n) + f(n) = O\big(g(n)\big).$$

Big O is multiplicative: if

$$f_1(n) = O\big(g_1(n)\big) \quad \text{and} \quad f_2(n) = O\big(g_2(n)\big)$$

then

$$f_1(n)f_2(n) = O\big(g_1(n)g_2(n)\big)$$

Big O simplification

Finding the big O notation of a function $f(x)$ is usually done using the following two rules:

- If $f(x)$ is a sum of more than one term, keep only the one which grows the fastest.
- If $f(x)$ is a product of more than one factor, remove any constant factor (doesn't depend on x).

Big O example

We consider the function:

$$f(n) = 10\log_2(n)n + 3n^2 + \log_2(n)$$

- If n is large then

$$3n^2 > 10\log_2(n)n > \log_2(n),$$

  so we keep only the second term $3n^2$.
- Next we remove the constant term ending up with:

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

Big O hierarchy

For any $a > 1$ and $p \in \mathbb{R}$,

$$n^p = O(a^n) \qquad \text{but} \qquad a^n \neq O(n^p).$$

For all $p, q \in \mathbb{R}$ such that $p < q$,

$$n^p = O(n^q) \qquad \text{but} \qquad n^q \neq O(n^p).$$

For any $a, p > 0$,

$$\log_a n = O(n^p) \qquad \text{but} \qquad n^p \neq O(\log_a n).$$

In particular, $O(\log_a n)$ is the same for all bases a, so the logarithm base is typically omitted inside a big O.

$O(1)$ is the class of all (asymptotically) bounded functions.

Example: Matrix addition

Input: Two $n \times n$ matrices A and B.
Output: The matrix $C = A + B$.

for i **from** 1 **to** n **do**
  for j **from** 1 **to** n **do**
    $C_{i,j} := A_{i,j} + B_{i,j}$;
  endfor
endfor

j loop body does $O(1)$ operations.
j loop repeats n times, so j loop does $O(n)$ operations.
i loop repeats n times, so i loop does $n \cdot O(n) = O(n^2)$ operations.
Algorithm complexity is $O(n^2)$.

Example: Matrix multiplication (naive algorithm)

Input: Two $n \times n$ matrices A and B.
Output: The matrix $C = AB$.

for i **from** 1 **to** n **do**
  for j **from** 1 **to** n **do**
    $C_{i,j} := 0$;
    for k **from** 1 **to** n **do**
      $C_{i,j} := C_{i,j} + A_{i,k} \cdot B_{k,j}$;
    endfor
  endfor
endfor

Algorithm complexity is $O(n^3)$.

Little O notation

Big O is a kind of "less than or equal to". The strict counterpart is little O:

$$f(n) = o\big(g(n)\big) \qquad \text{(read '}f(n) \text{ is little O of } g(n)\text{')}$$

means

for every $\varepsilon > 0$ *there is some* N *such that* $\big|f(n)\big| \leqslant \varepsilon \big|g(n)\big|$ *for all* $n > N$.

If $g(n)$ is nonzero for sufficiently large n, the above is equivalent to

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0.$$

Smaller is better?

The general sentiment is that an algorithm is better the lower its asymptotic complexity, or more formally:

> *If algorithm A has complexity $f(n)$ and algorithm B has complexity $g(n)$, where $f(n) = o\big(g(n)\big)$, then A is better than B.*

Caveats:

- To achieve lower complexity, the algorithm often needs to be more complicated, and may thus be harder to implement correctly.
- The O notations hide constants, which *may* make a difference for problems of the size you're interested in.

Algorithms for matrix multiplication

- Naive matrix multiplication is $O(n^3)$.
- Strassen's matrix multiplication algorithm (1969) is $O(n^{2.81})$.
- From 1980 and forth, various $o(n^{2.5})$ algorithms have been presented (Coppersmith–Winograd and others).

In general, the lower the exponent, the higher the constant factor that is hidden by the big O notation (though not according to a simple pattern). Strassen's $O(n^{2.81})$ algorithm perform fewer operations than the naive one for $n > 100$, if implemented well.
The various $o(n^{2.5})$ algorithms all have gigantic constants, and are inferior for all matrices of managable size.

Flavours of complexity

The algorithm complexity is a bound, but what is it a bound *of*?

- If nothing else is said, it should be a bound for all possible outcomes, even the most unfavourable ones. This is called worst-case complexity.
- Average-case complexity is instead a bound for the expected outcome for random input of the given size.
- Amortised complexity is a bound for the average complexity of an operation if it is repeated a large number of times. This can be appropriate for operations that are part of a larger algorithm.

Example: Exponential growth of list

An *unbounded length list* can be implemented using:

- a pointer to a dynamically allocated vector of items,
- a counter N for the number of elements in the list,
- a counter K for the number of elements in the vector.

If $N < K$, then appending an element to the list is worst-case $O(1)$. But if $N = K$ (the vector is full), one must first allocate a new, larger vector and copy all data from the old one to the new before adding the new element. This is an $O(N)$ operation.

If the vector size is grown by a constant factor $c > 1$ each time it is resized, then it is possible to achieve $O(1)$ *amortised* complexity for appending.

Concrete complexity classes

One classification refers to concrete complexity estimates:

constant	$O(1)$
logarithmic	$O(\log n)$
linear	$O(n)$
quadratic	$O(n^2)$
cubic	$O(n^3)$
polynomial	$O(n^d)$ for some constant d
exponential	$O(a^n)$ for some constant a
doubly exponential	$O(a^{a^n})$

Sub- and super- are sometimes used as prefixes to the above to mean "less than" and "more than" respectively, sometimes with an implied "slightly".
$n!$ and n^n are both superexponential, but only slightly so, since they are both $O(2^{n \log n})$.

Anything less than linear complexity means you'll have to accomplish the task without even looking at all the input! This is rare, but happens, and may be dependant upon the computation model.

For larger time complexities (polynomial and up), the exact model usually doesn't matter, because the different models can emulate each other well enough.

The exact format of the input can matter greatly in low complexity settings, but is less important in higher complexity, as a preprocessing step translating to the preferred format tends to contribute only a lower order term to the total complexity.

Note that the n in matrix algorithm complexity estimates tends to be the matrix side rather than the formal input size. A general $n \times n$ matrix requires at least Cn^2 bits to encode, so in that sense matrix addition is a linear time operation.

Theoretical complexity classes

There is also a more theoretical hierarchy of complexity classes, which refers more to families of computational *problems* and the complexity of an optimal algorithm for solving these.

P is the class of all problems for which the output is a boolean (yes/no problems) and for which there is a polynomial algorithm.

NP is the class of all problems with boolean output for which the corresponding verification problem (checking someone's substantiated claim that the answer is 'yes') is in P.

A typical example of an NP problem would be to ask whether a system of inequalities has a solution. If the answer is 'yes', then the claimant can substantiate this claim by giving that solution; verifying that something is a solution can easily be done in polynomial time.

Surprisingly, the majority of all *NP* problems for which no polynomial algorithm is known are *NP*-complete: any other *NP* problem can be transformed to an instance of that problem. Hence they are all "equally difficult".

All known algorithms for *NP*-complete problems have exponential time complexity.

It is not a severe restriction that *NP* problems only answer yes/no questions. A bit string of any length can be calculated through a series of yes/no questions asking whether the kth bit is 1.