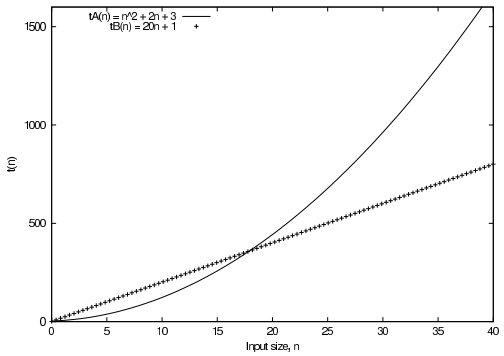
Lecture 27:
Asymptotic Analysis

Aims:

- To look at the idea of the growth rate of a function;
- To look at big-Oh notation.

27.1 Growth Rates

- Suppose algorithm A's worst-case time complexity $t_A(n) =_{\text{def}} n^2+2n+3$ and algorithm B's worst-case time complexity $t_B(n) =_{\text{def}} 20n+1$. Which is the faster algorithm?



  For inputs of size 0, B is faster (1 step compared to 3 steps). Then for inputs whose size is greater than 0 but less than 18, A is faster, and then for inputs of 18 or more B is faster.

  It all seems less than clear-cut.

- The software practitioner in industry would probably want to think about what size of inputs the algorithm will *typically* face in practice. If the size of the inputs will usually be < 18, then s/he might choose to code up and use algorithm A; but if the size of the inputs will often be ≥ 18, then s/he might choose to code up and use algorithm B. (Of course, s/he needs to be aware of all the simplifications that our analysis has made so far, e.g. we are considering worst-cases and we are counting steps rather than using actual timings.)

- But the theorist often wants to talk in more general terms. Despite the fact that these graphs cross twice, the theorist may want to say something about which algorithm is better in general.

  How can we make such a statement?

  - Large input sizes are what matter most. If the problem you are trying to solve is to sort someone's personal address book of, say, 10–100 entries, then it hardly matters which sorting algorithm you choose. But if the problem is to sort national telephone directories of, say, between 4 and 60 million entries, then it does matter which algorithm you choose (especially if you want to sort the directories repeatedly at the request of on-line users).
  - In the example, from $n = 18$ onwards, B is better and its advantage over A gets ever bigger. The theorist ignores the fact that A is better for most small values.

- So, what matters is how well an algorithm 'scales up' to larger problem instances.
  - To determine how well an algorithm 'scales up', we use the *growth rate* of the time complexity function.
  - If A's growth rate is greater than B's then B is a better algorithm than A.
  - A may (or may not) be faster than B on 'small'-sized inputs. But, for 'sufficiently large'-sized inputs, B will be more efficient, and its advantage will get bigger as the input size grows.
  - How do you find out the growth rate of a function?
  - Consider a function $t(n)$ expressing, e.g., the worst-case time complexity of an algorithm.
  - As n grows large, some term in a function may begin to dominate the other terms.
  - For example, for $t_A(n) =_{\text{def}} n^2 + 2n + 3$, as n grows larger, n^2 grows very much larger than $2n$; the $2n$ and the 3 become ever more irrelevant.
  - For example, for $t_B(n) =_{\text{def}} 20n + 1$, as n grows larger, the $20n$ dominates and the 1 becomes ever more irrelevant.
  - So we concentrate on the dominating term. For A this is n^2; for B, this is $20n$.
  - Because we are considering what happens as n grows ever bigger, towards infinity, we call this the *asymptotic* behaviour of the function or algorithm.
  - We go even further: we ignore the coefficient of the dominating term.
  - So, we say A grows proportionally to n^2 and B grows proportionally to n.
  - Comparing these: B is the better algorithm.
  - Of course, this has made our analysis even more rough-and-ready.

- Maybe a concrete example will help to convince you that ignoring coefficients and lower-order terms makes quite reliable judgements.
  - The complexity of linear search grows proportionally to n.
  - The complexity of binary search grows proportionally to $\log n$.
  - $\log n$ is better than n.

n	$\log n$
10	3
100	6
1000	9
a million	19
a billion	29
a billion billions	59

  - You can see that coefficients and lower-order terms are not really important to the different growth rates!

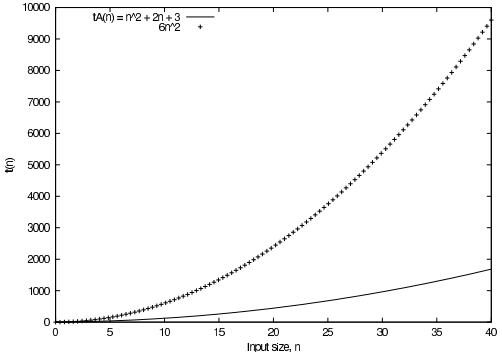
27.2 Big-Oh Notation

27.2.1 Informal Discussion

- To talk about growth rates, we often use *"big-Oh" notation*.
- E.g. we might say the worst-case time complexity of an algorithm is $O(n^2)$
  - "big-Oh of n squared"
  - "Oh of n squared"

  It just means that its growth rate is n^2.
- If we say that an algorithm's worst-case time complexity is $O(n^2)$, we mean that

  there are positive constants c and n_0 such that for all $n \geq n_0$, we have $t(n) \leq cn^2$.
- In other words, $t(n)$ is $O(n^2)$ means
  - we can find two positive numbers, c and n_0,
  - we plot $t(n)$
  - we plot cn^2
  - for values of n from n_0 onwards, the line for cn^2 will be higher than or the same as that for $t(n)$
  - it won't necessarily be higher or equal for 'small' values of n ($n < n_0$) but we don't worry about these cases.
- For example, suppose $t_A(n) =_{\text{def}} n^2 + 2n + 3$. Then $t_a(n)$ is $O(n^2)$.



27.2.2 The Formal Definition

- More formally and more generally,

  $t(n)$ is $O(f(n))$ if there are positive constants n_0 and c such that, for all $n \geq n_0$, $t(n) \leq c \times f(n)$
- Examples

  - $n^2 + 2n + 3$ is $O(n^2)$ (e.g. $n_0 = 1$ and $c = 6$)
  - $3n^3 + 2n^2$ is $O(n^3)$ (e.g. $n_0 = 1$ and $c = 5$)
  - $n^2 + 2n + 1$ is $O(n^2)$ (e.g. $n_0 = 1$ and $c = 4$)
  - $3n \log n + n + 2$ is $O(n \log n)$ (e.g. $n_0 = 1$ and $c = 6$)
  - $2^n + n^6 + 17$ is $O(2^n)$ (e.g. $n_0 = 5$ and $c = 4$)

27.2.3 Proofs

- To prove, e.g., that $n^2 + 2n + 3$ is $O(n^2)$, you must come up with the two numbers n_0 and c and show that for all $n \geq n_0$, $t_A(n) \leq cn^2$.

  We can show that

  $$n^2 + 2n + 3 \leq n^2 + 2n^2 + 3n^2 = 6n^2$$

  We compare corresponding terms.
  - Is $n^2 \leq n^2$? Yes, for all $n \geq 0$.
  - Is $2n \leq 2n^2$? Yes, for all $n \geq 0$.
  - Is $3 \leq 3n^2$? Yes, for all $n \geq 1$.

  Therefore we have shown that $n^2 + 2n + 3 \leq cn^2$ for all $n \geq n_0$ with $c = 6$ and $n_0 = 1$.
- We did not necessarily find the smallest value for c that would prove this, but at least it was quite easy to come up with.

27.2.4 Disproof

- To prove, e.g., that $2n^3$ is *not* $O(4n^2)$.

  Assume $2n^3$ is $O(4n^2)$. Then, from the definition, there must exist constants c and n_0 such that

  $$\begin{array}{rcll} 2n^3 & \leq & c \times 4n^2 & \text{for all } n \geq n_0. \\ \frac{2n^3}{4n^2} & \leq & c & \text{for all } n \geq n_0. \\ \frac{n}{2} & \leq & c & \text{for all } n \geq n_0. \end{array}$$

  Or, if you prefer, $c \geq \frac{n}{2}$ for all $n \geq n_0$.

  But, this is a contradiction: $\frac{n}{2}$ can get arbitrarily large as n gets larger, so no constant can exceed $\frac{n}{2}$ for all n. Therefore $2n^3$ is not $O(4n^2)$.

27.2.5 Tightness

- The definition does not say how 'tight' the approximation should be.
- So, for example, $3n + 2$ is $O(n)$ but it is also $O(n^2)$, $O(n^{2.5})$, $O(n \log n)$, $O(2^n)$, and so on.
- If you are describing an algorithm's complexity using only big-Oh notation, then you are expected to choose a function that is as informative as possible.
- I.e. $f(n)$ should be as small a function of n as you can reasonably come up with for which $t(n) \leq c \times f(n)$.

27.2.6 Simplifying Big-Oh Notation

- People would think it poor practice if you were to say that an algorithm's complexity is, e.g., $O(3n)$ or $O(3n+2)$, even though there's nothing in the formal definition to disallow this. Instead, in this example, you would just say $O(n)$.
- You can simplify big-Oh expressions using the following rules:
  - Low order terms don't matter, so strike them out. E.g.:
    $$O(3n+2) = O(3n)$$
  - Coefficients don't matter, so strike them out. E.g.:
    $$O(3n) = O(n)$$
- So people restrict themselves to some common, simple functions:

$O(1)$	$O(2^n)$
$O(\log n)$	$O(n!)$
$O(n)$	$O(n^n)$
$O(n \log n)$	
$O(n^2)$	
$O(n^3)$	

- There is an inclusion relationship:
  $$O(1) \subset O(\log n) \subset O(n) \subset O(n \log n) \subset O(n^2) \subset O(n^3) \subset O(2^n) \subset O(n!) \subset O(n^n)$$
- And, to repeat the point about tightness, you are expected to describe your algorithm using the tightest set possible.

27.2.7 Computing Big-Oh Expressions

- The impression given so far is
  - first we compute $t(n)$ as per previous lectures;
  - then we work out the growth rate from $t(n)$ as above.
- But we can short-cut this. We do not need to compute $t(n)$. We can compute the growth rate directly from the algorithm.
- This is one of the reasons why this is all worth doing. The growth rate can be much more quickly and easily computed than can $t(n)$.
- So, given an algorithm, here's some guidance for quickly and directly obtaining the growth rate in big-Oh notation.

  Assignments: These are $O(1)$ (similarly, **return** commands)

  Sequences: If there are 2 commands in sequence and they are $O(f_1(n))$ and $O(f_2(n))$, then the growth rate of the sequence is $O(\max(f_1(n), f_2(n)))$.

  An advanced point is that this assumes that we can compare $f(n)$ and $g(n)$ to select the maximum. Sometimes $f(n)$ and $g(n)$ will be incommensurate, in which case the maximum cannot be given. There's no need to worry about this possibility in CS2205.

One-armed conditionals:
- The test is $O(1)$.
- Suppose the branch is $O(f(n))$.

Then the one-armed conditional as a whole is also $O(f(n))$.

Two-armed conditionals:
- The test is $O(1)$.
- Suppose the **if** branch is $O(f_1(n))$.
- Suppose the **else** branch is $O(f_2(n))$.

Then the two-armed conditional as a whole is $O(\max(f_1(n), f_2(n)))$.

Unbounded iterations:
- Let $g(n)$ be an upper bound on the number of times we may go around the loop.
- The test is $O(1)$.
- Suppose the body is $O(f(n))$.

Then the loop as a whole is $O(g(n) \times f(n))$, which might then simplify.

Bounded iterations: The reasoning is the same: the loop as a whole is $O(g(n) \times f(n))$. (Initialising the counter is $O(1)$; incrementing the counter adds $O(1)$ to the body of the loop.)

- This looks nasty but it's easy for simple algorithms. With experience you learn to quickly ignore the bits that cost $O(1)$ and concentrate on the loops.
- Here's an example.

Algorithm: PREFIXAVERAGES1(a)

```
create array b[1...n]
for i := 1 upto n
{   sum := 0.0;
    for j := 1 upto i
    {   sum := sum + a[j];
    }
    b[i] := sum/i;
}
return b;
```

This has $O(n^2)$ worst-case time complexity.

- Creation and initialisation of the array is $O(n)$.
- The inner loop:
  - Loop body is $O(1)$.
  - Loop is executed i times, which in the worst case is n times.
  - So the loop as a whole is $O(n \times 1) = O(n)$.
- The outer loop:
  - The loop body comprises an assignment ($O(1)$), the inner loop ($O(n)$) and another assignment ($O(1)$) in sequence. It is therefore $O(\max(1, n, 1)) = O(n)$.
  - Loop is executed n times.

    - So the loop as a whole is $O(n \times n) = O(n^2)$.
- The program as a whole comprises the array initialisation ($O(n)$) and the loop ($O(n^2)$) in sequence. So we have $O(\max(n, n^2)) = O(n^2)$.

Class Exercise

What is the worst-case time complexity of this one:

Algorithm: PREFIXAVERAGES2(a)

```
create array b[1...n]
sum := 0.0;
for i := 1 upto n
{    sum := sum + a[i];
     b[i] := sum/i;
}
return b;
```

Acknowledgements

The idea of showing a comparison between linear search and binary search comes from [Har92]. A lot of the material on big-Oh notation is based on [AU92].

References

[AU92] A. V. Aho and J. D. Ullman. *Foundations of Computer Science*. W.H. Freeman, 1992.

[Har92] D. Harel. *Algorithmics: The Spirit of Computing*. Addison-Wesley, 2nd edition, 1992.