Cosc 241
Programming and Problem Solving
Lecture 6 (12/3/20)
Analysis of recursive algorithms

Michael Albert
michael.albert@cs.otago.ac.nz

UNIVERSITY of OTAGO
SAPERE AUDE
Te Whare Wānanga o Otāgo
NEW ZEALAND

Factorial

Factorial(n):
  if $n = 1$ **then**
    return n
  else
    return $n \times$ Factorial($n - 1$)
  end if

How many multiplications, $f(n)$, are required?

If $n = 1$, $f(n) = 0$. If $n > 1$ then $f(n) = 1 + f(n-1)$. So $f(n) = n - 1$, in particular $f(n) = O(n)$.

General rule: “constant plus previous case” gives $O(n)$.

Greatest common divisor

- How can we compute the greatest common divisor of two positive integers a and b (suppose that $a \leq b$)?
- First case, a is a divisor of b, then the answer is a.
- Otherwise, the answer is the greatest common divisor of a and the remainder when b is divided by a.
- Can collapse first case into second by remembering that $\gcd(0, b) = b$ for all b.

GCD(a, b):

if $a = 0$ **then**
  return b
end if
return GCD($b \,\%\, a$, a)

Sample computation

$$\begin{aligned}\gcd(15, 24) &= \gcd(9, 15)\\ &= \gcd(6, 9)\\ &= \gcd(3, 6)\\ &= \gcd(0, 3)\\ &= 3\end{aligned}$$

- How many steps (i.e., recursive calls) might this require in the worst case?
- As we see above, we could well have $b \,\%\, a$ quite large relative to a.
- But, if it is $> a/2$ then the next remainder is $< a/2$, while if is $\leq a/2$ then the next remainder is smaller still.
- The first argument is reduced by a factor of at least 2 every two steps.

How often can we divide by two?

- If $a < 2^n$, then after n integer divisions by two, we will get 0.
- So if we take the logarithm (base 2) of a and round up – that many divisions kills us.
- In other words, the number of calls in GCD is at worst $2 \log a$. We can actually do a bit better than this but don't care because constants.
- Its complexity is $O(\log a)$.
- General rule: any recursive algorithm where the size reduces by a constant factor (greater than 1) in one call has complexity $O(\log n)$ (and that's good!)

Recursive Fibonacci

Fib(n):

if $n \leq 1$ **then**
  return 1
end if
return Fib($n-1$) + Fib($n-2$)

- How much work to compute Fib(n)? Call the amount of work, $f(n)$.
- One test, one arithmetic operation, one call to Fib($n-1$) and one to Fib($n-2$), so:

$$f(n) = 2 + f(n-1) + f(n-2)$$

- The amount of work done behaves like the Fibonacci numbers themselves, and they grow exponentially.
- General rule: two (or more) recursive calls where the size reduces by a constant amount leads to exponential complexity (bad!)

Computing powers

Problem: Evaluate a^n

- A naive recursive (or iterative) algorithm uses $a^1 = a$, $a^n = a \times a^{n-1}$ and has complexity $O(n)$ (in fact, does exactly $n - 1$ multiplications).
- Can we do better?
- First thing to observe is that if n is even, we frequently can, e.g.,
$$a^6 = (a^2)^3 = (a \times a)^3$$
- Since we can first compute a^2 (one multiplication), then compute the third power *of that* (two multiplications), we use three multiplications instead of five.
- But what about the odd case?

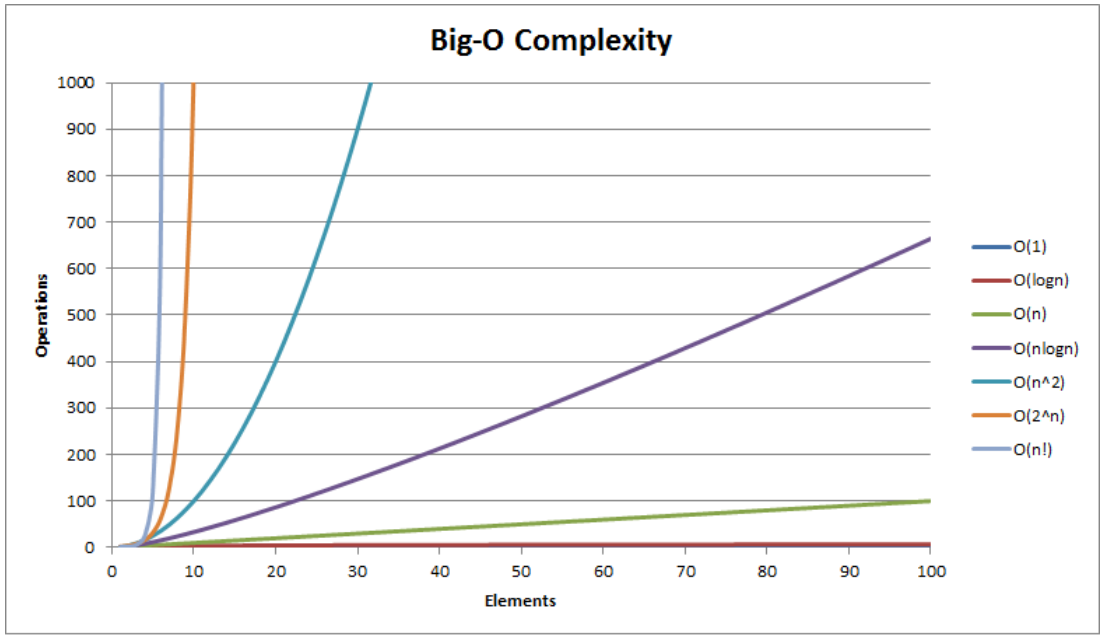
Clever powering

Power(a, n):

if $n = 0$ **then**
  return 1
end if
$p \leftarrow$ Power$(a \times a, n/2)$ (integer division)
if $n \,\%\, 2 = 1$ **then**
  $p \leftarrow a \times p$
end if
return p

In the worst case we do one test, one multiplication ($a \times a$), one call reducing n by a factor of 2, one more test, and one more multiplication. By our general rules, that's $O(\log n)$.

Big-Oh algebra



- The O can absorb a lot of stuff for instance:
  - if $f(n) = O(g(n))$ then $100f(n) + 3g(n) = O(g(n))$,
  - if $f(n) = O(g(n))$ and $g(n) = O(h(n))$ then $f(n) = O(h(n))$
- The important thing is to remember to read $f(n) = O(g(n))$ as "(some multiple of) g grows at least as quickly as f", and then use common sense.