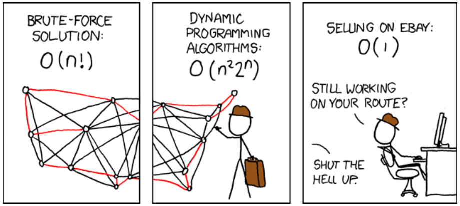
Cosc 241
Programming and Problem Solving
Lecture 5 (9/3/20)
Analysis of Algorithms (1)

Keywords: Big-Oh

Michael Albert

michael.albert@cs.otago.ac.nz



http://xkcd.com/399/

Algorithmic efficiency

- There are two principal types of algorithmic efficiency that are important: time efficiency, and space (memory) efficiency.
- Of these, the first is more commonly critical, and will be the one which we discuss in COSC241.
- We seek a quantitative notion of efficiency which nevertheless ignores insignificant details like clock rate, machine load, or compiler optimisations.
- The key insight is that most problems have a natural size parameter (usually denoted n) and the critical issue is how the performance of the algorithm changes as n grows.

Some possibilities

- The algorithm might run in (roughly) the same amount of time, no matter what *n* is.
- When *n* doubles, the algorithm might take (roughly) some constant number of extra steps.
- When *n* doubles, the algorithm might take (roughly) twice as long.
- When *n* increases by some constant amount, the algorithm might take (roughly) twice as long.
- How can we capture all of these possiblities, including the "(roughly)" part?

Discussion

Imagine that we could agree on exactly what constituted a "basic step". We might then find that the number of steps required by an algorithm was:

$$17 + 5n + 3n^2 \lceil \log n \rceil + 7 \times 2^n.$$

The actual time required (on a particular computer, under identical load conditions) would be some constant multiple (steps per second) of this.
But the important thing is that in the expression above, there is one term that is much larger than all the others (at least for any significantly large value of *n*).

A working compromise

- Ignore inessential pieces of the exact formula.
- In fact, don’t even worry about computing the exact formula, concentrate on producing upper bounds that are correct up to a constant factor.
- Assume that *n* is going to be large.

Big-Oh notation

- Suppose that $f(n)$ and $g(n)$ are two functions of n.
- We write
$$f(n) = O(g(n))$$
if, for some constant A, $f(n) \leq Ag(n)$ for all sufficiently large n.
- That is, up to some constant factor, $g(n)$ provides an upper bound for $f(n)$.
- Normally, $f(n)$ is some complicated, perhaps not exactly known, expression and $g(n)$ is some simplified version.
- If $f(n) = O(g(n))$ then essentially, $f(n)$ grows no more quickly than $g(n)$ does.

A sample computation

We have

$$3 + 5n^2 + 7n^3 = O(n^3)$$

because for all positive integers n:

$$3 + 5n^2 + 7n^3 \leq 3n^3 + 5n^3 + 7n^3 = 15n^3.$$

By a similar argument, any polynomial (sum of fixed powers of n) is O of the highest power that occurs.

Many O computations/verifications are a bit more complicated than this, but none ever need any advanced maths. The fact that you can ignore all the small fiddly bits actually makes things easier!

Try this (2 minutes)

True or false?

- $3 + 5n + 18n^2 = O(n)$
- $3 + 5n + 18n^2 = O(n^2)$
- $3 + 5n + 18n^2 = O(n^3)$
- $3^n = O(n^3)$
- $n^3 = O(3^n)$

The scale of growth rates

- One of the main reasons for using the O notation is that it provides a scale of common growth rates for functions (and the efficiency of algorithms).
- From least quickly growing (= most efficient!), to most quickly growing:

$$1,\ \log n,\ \sqrt{n},\ n,\ n\log n,\ n^2,\ n^3,\ 2^n.$$

- Algorithms whose time complexities have growth rates near the beginning of this list are to be preferred.
- In practice, algorithms with growth rates like 2^n are generally impractical except for very small values of n.

Practicalities

You will learn to recognize certain common scenarios which may or may not contribute to the essential time complexity of an algorithm:

- pre and post-processing usually take a constant amount of time, $O(1)$, and can often be ignored;
- a simple for loop from 0 to n (with no internal loops), contributes $O(n)$ (linear complexity);
- a nested loop of the same type (or bounded by the first loop parameter), gives $O(n^2)$ (quadratic complexity);
- a loop in which the controlling parameter is divided by two at each step (and which terminates when it reaches 1), gives $O(\log n)$ (logarithmic complexity);
- the "divide and conquer" paradigm (later!) which breaks the problem into two instances of size $n/2$ which must then be combined in linear time gives $O(n \log n)$.

Square testing revisited

```
Algorithm 0
  $i \leftarrow 1$
  while $i \leq n$ do
    if $i^2 = n$ then
      Answer “yes”, halt.
    end if
    $i \leftarrow i + 1$
  end while
  Answer “no”, halt.
```

We have a single loop, executed n times (in the worst case). So the complexity is $O(n)$.

Algorithm 1

```
$i \leftarrow 1$
while $i^2 \leq n$ do
   if $i^2 = n$ then
      Answer “yes”, halt.
   end if
   $i \leftarrow i + 1$
end while
Answer “no”, halt.
```

Now we do (some constant) $\times \sqrt{n}$ operations whether or not n is a perfect square, i.e. the complexity is $O(\sqrt{n})$.

Algorithm 2

```
$i \leftarrow 1$
**while** $i^2 \leq n$ **do**
  $i \leftarrow 2 \times i$
**end while**
$high \leftarrow i,\ low \leftarrow i/2$
If $low^2 = n$ then answer yes and halt
**while** $high - low > 1$ **do**
  $mid \leftarrow (high + low)/2$
  **if** $mid^2 = n$ **then**
    Answer yes and halt
  **else if** $mid^2 < n$ **then**
    $low \leftarrow mid$
  **else**
    $high \leftarrow mid$
  **end if**
**end while**
Answer no and halt
```

A bit more complicated!

Analysis of Algorithm 2

- In the first loop, we keep doubling i until it's bigger than $\sqrt{n}$. That is, until we have:

  $$2^k = i > \sqrt{n}$$

  which means

  $$k > \log n/2.$$

  So that loop is $O(\log n)$.
- Now we have *high* and *low* which differ by $i/2$, i.e., by at most $\sqrt{n}$.
- In the next loop, we calculate their average, and it replaces one or the other of them (or we terminate).
- So their difference goes down by a factor of two each time through that loop.
- In the worst case, that's another $O(\log n)$.
- And so the whole thing is $O(\log n)$.