Cosc 241
Programming and Problem Solving
Lecture 3 (2/3/2020)
Algorithms

Michael Albert

michael.albert@cs.otago.ac.nz

Keywords: algorithm, pseudocode, efficiency

UNIVERSITY of OTAGO

Te Whare Wānanga o Otāgo

NEW ZEALAND

Lecture outline

- What is an algorithm?
  - History
  - Examples
- Measuring efficiency/cost

What is an algorithm?

- An algorithm is a description of a series of basic steps that achieve some specified result.
- The description of each step must be precise.
- The sequence of steps followed must be rigorously and unambiguously defined, but may depend on available information.
- An algorithm may have some input values.
- An algorithm may (in practice, will) have some outputs, or side effects that bring about the specified result.
- A useful algorithm is guaranteed to terminate in a finite number of steps!

What is a basic step?

- It depends!
- An algorithm is a description of a procedure that is intended to be carried out.
- The notion of a basic step depends on who/what is going to be executing the algorithm.
- If it is a computer, then a basic step might be as basic as: "add *a* to *b* and store the result in *c*" (or as complicated as "Open a new window in which the user can enter text").
- If it is a human, then it might well depend on his/her skill or experience ("make 500ml of Béchamel sauce" or "Place a heavy saucepan over low heat . . . ").

History

- We have a long tradition of telling people what to do and how to do it!
- In ancient India, China, Sumeria, Egypt, Greece various algorithms relating mostly to geometry (and surveying) and arithmetic.
- The Persian mathematician al-Khwarizmi (c. 800 CE) described many arithmetic, algebraic, and geometric algorithms. The word algorithm is derived from the Latin translation of his name.
- Not until the 1930's was the notion of algorithm formalised (Turing, Church) and the development of electronic computing has led to its intensive study.

Try this

Describe to your neighbour an algorithm for one of the following objectives, then have them describe an algorithm for another one:

- Get to the Dunedin railway station from here.
- Make a paper airplane.
- Find the meaning of the word “sesquipedalian”.
- Get an A in COSC241.

Example

Problem: Given a positive integer n, determine whether or not it is a perfect square.

Algorithm 0: Compute the square of each successive positive integer from 1 to n. If the answer is ever n, answer “Yes” (and halt). Otherwise, answer “No” (and halt).

That’s a high level description in English.

Example

Problem: Given a positive integer n, determine whether or not it is a perfect square.

Algorithm 0

```
i ← 1
while i ≤ n do
    if i² = n then
        Answer "yes", halt.
    end if
    i ← i + 1
end while
Answer "no", halt.
```

From algorithms to programs

- The more abstract the language we use to describe an algorithm, the closer it is to a computer program.
- So why bother? Why not just write a program that expresses the algorithm?
- Because a program carries a lot of baggage (boilerplate, exception handling, input/output) which often obscures the essential nature of the algorithm.
- Still, it's a good habit to try and write programs whose underlying algorithms shine through.

Efficiency and cost

- For computing there are two things we need to worry about when thinking about implementing an algorithm: time and space.
- For time, we usually measure the number of basic steps (e.g., single arithmetic operations) required by an algorithm based on some measure of the size of its input.
- For space, we consider the amount of memory required by the algorithm as its “working area” again, as it relates to the size of the input.
- Initially we don’t need to worry too much about details – ballpark estimates will allow us to compare two competing algorithms.
- We will see how to do this in general next week.

How efficient is Algorithm 0?

- In each iteration of the loop we do one multiplication, two comparisons, and one increment.
- Outside of the loop we only do an assignment.
- The loop body is executed either n times (if n is not a perfect square), or $\sqrt{n}$ times (it if is).
- So, the total number of operations in the worst case, is something like $4n$ (or maybe multiplications cost more than the other operations?)
- In analysing efficiency, we always focus on the worst case.

Improving Algorithm 0

- Algorithm 0 is much more efficient when n is a perfect square than when it isn't.
- Is there any way we can arrange that the two cases aren't so different?
- Yes! If we notice that the values of i^2 always increase as i does. So, once we get past n we can be sure that we'll never hit it.
- In words: starting from 1 compute the squares in succession; if you ever hit n then answer yes and halt, otherwise, as soon as you pass n answer no and halt.

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.

Are we happy?

- The keen algorithmicist (I just made that word up) is never happy, unless convinced that an algorithm is as efficient as it could possibly be.
- But how do you know?
- Look for places where there's room for improvement, or imagine being able to guess really well.
- If we could guess the integer k at, or just below, $\sqrt{n}$, then just by looking at k^2 and $(k+1)^2$ we could prove whether or not n was a perfect square.
- So in the "really good guessing" model we might hope for a constant number of operations.
- More realistically, the obvious place where we have inefficiency is in $i \leftarrow i + 1$. If n is large, then we waste a lot of time computing small squares.

A new idea

- Start as usual at 1, and compute squares. Each time the result is less than *n*, double the number you're squaring until the result is larger than *n*.
- Now the square root of *n* lies between the final value you squared (which was too big), and half that value.
- You could now just scan through that interval (as in Algorithm 1 say), but having had this idea you'll probably think of
- looking at the midpoint! This is either too big, too small, or just right.
- In any case you either have a new set of candidate values which is half as big as before, or you're done.
- Trust me, the efficiency is now (some constant) $\times \log n$ and that's a lot better!

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

Counting loops

n	Algorithm 0	Algorithm 1	Algorithm 2
10	10	3	3
100	10	10	6
1000	1000	31	9
10000	100	100	11
100000	100000	316	17
1000000	1000	1000	16
10000000	1000000	3162	23
100000000	10000	10000	23
1000000000	1000000000	31622	29
10000000000	100000	100000	28
100000000000	Too long!	316227	37