Lecture 8
Algorithms and complexity

COS10003 Computer Logic and Essentials (Hawthorn)

SWINBURNE UNIVERSITY OF TECHNOLOGY

Semester 1 2021

Today

1. Review
2. Complexity
3. Recurrence
4. Induction

How algorithms can be compared

The different time complexities

How recursion is beneficial

There are two main methods of representing algorithms:

- Flowcharts
- Pseudocode

Both show the logical connectivity of the algorithm and suggest how it could be implemented in a programming language.

Programming concepts

- **Sequence**: getting steps in the right order
- **Selection**: choosing between two paths
- **Iteration**: repeating the same instructions many times
- **Invocation**: moving code that is called frequently into a function or procedure

Selection: if-then-else

What happens next in an algorithm could depend on the outcome of one or more conditions.

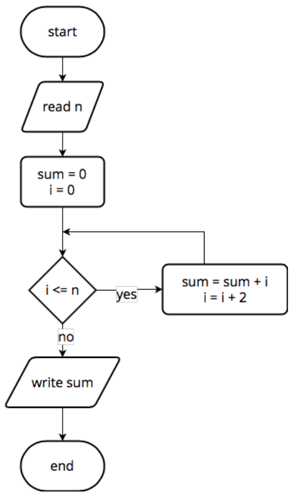
```
1: if $condition1$ then
2:     steps if condition 1 is true
3: else if $condition2$ then
4:     steps if condition 2 is true
5:     ...
6: else
7:     steps if all other conditions are false
8: end if
```

Iteration

- Iteration logic requires control structures involving loops.
- A feature of all loops is that they must contain initiating and stopping.
- Failure to satisfy the former means that no iteration will occur whereas omitting a stopping condition will result in an infinite loop.
- Let's try finding the the sum of even numbers up to and including n.

Pre-test

```
Algorithm 1 $\sum_{0,2,4...}^{n} i$
Require: n > 0
    read n
    sum = 0
    i = 0
    while i ≤ n do
        sum = sum + i
        i = i + 2
    end while
    print sum
```



Invocation

Invocation is using another program/algorithm in your program/algorithm in order to solve a specific (sub-)problem.

We can indicate invocation in flowcharts with a circle; this indicates that we should move to another flowchart.

What is an algorithm?

An algorithm is an well-defined computational problem that takes some values, or a set of values, as input and produces some value, or a set of values, as output. An algorithm is thus a sequence of computational steps that transform the input into the output.

Cormen, Leiserson, Rivest and Stein, chapter 1 of Introduction to Algorithms

What we want

We want algorithms that:

- produce a correct solution to the problem
- use computational resources efficiently while doing so

Cormen, chapter 1 of Algorithms Unlocked

Not all equal

- Not all algorithms are equal.
- if we have the choice between two algorithms that solve the problem with the same accuracy, we would prefer the more efficient or faster solution.
- There are two approaches to determining this:
  - Running the algorithms and measuring performance
  - Undertaking a mathematical analysis

Greatest common divisor 0.1

The greatest common divisor (gcd) of two or more integers, which are not all zero, is the largest positive integer that divides each of the integers (Wikipedia:GCD). Let's use a and b.

- For each number i from 1 to $a/2$: add i to a list if a is divisble by i
- For each number j from 1 to $b/2$: add j to another list if b is divisble by j
- Take both lists, and find the largest number that is common to both.

Greatest common divisor 0.2

1. Repeatedly subtract a from b until $b < a$
2. Swap the values of a and b
3. Repeat steps 1 and 2 until $b = 0$
4. The GCD of a and b is a

Let’s count some code

Algorithm 2 Quadratic formula

```
read a, b, c
d = b*b – 4ac
if d is negative then
    print ”no real solutions”
else if d = 0 then
    x = -b / 2a
    print ”unique solution:”, x
else
    x1 = (-b + sqrt(d)) / 2a
    x2 = (-b - sqrt(d)) / 2a
    print ”two solutions: ”, x1, x2
end if
```

- Worst case is the time for finding d plus the slowest branch.
- Count each operation as 1: approximately 7 + 14 = 21 steps.
- Constant based on number and size of inputs.

Let's count some code

```
Algorithm 3 $\sum_{0,2,4...}^{n} i$
Require: $n > 0$
    read n
    sum = 0
    i = 0
    while i $\leq$ n do
        sum = sum + i
        i = i + 2
    end while
    print sum
```

- Time for reading and setting initial values, plus writing the sum at the end, plus doing some assignments in the loop.
- Let's count each operation as 1, so will arrive at a constant value + n/2 * another constant value for the loop.
- T(input) = approx. 3 + 4 * n/2 + 1 = 4 + 2n
- Anything with a single loop that runs n times will need n steps; this will increase with n

Analysing algorithms

We can classify algorithms in terms of their running time based on a primary parameter (N) such as the number of data items processed, the degree of a polynomial, the size of a file to be searched, the number of bits in the input etc., which has the dominant role in affecting the overall execution time.

Playtime

Relative speeds

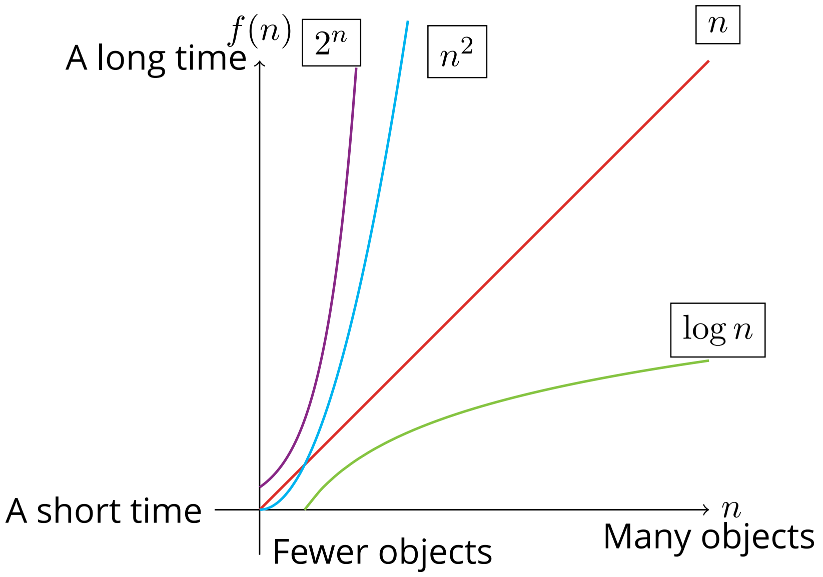
Without a basket
For one extra object beyond N, the algorithm takes $2N$ extra steps.
The time taken for N objects is in the order of N^2 steps
($2(n + (n-1) + (n-2) + (n-3))$... etc.
= $n(n+1)$)
The space taken for N objects is 1.

With a basket
For one extra object beyond N, the algorithm takes 1 extra step.
The time taken for N objects is in the order of N steps.
The space taken for N objects is N.

Big O notation

The running time of an algorithm is $\Theta(f(n))$. This means that the running time of the algorithm in all cases follows the same function $f(n)$.
However we specify this very broadly.

Order of growth



Big O notation – upper bound

Sometimes algorithm runtimes differ with different inputs.
We generally talk about an upper bound, denoted by O.
The formal definition is:

Definition of O

A function $f(N)$ is $O(g(N))$ if
for some constant c and
for values of N greater than some value n_0 then
$f(N) \leq cg(N)$

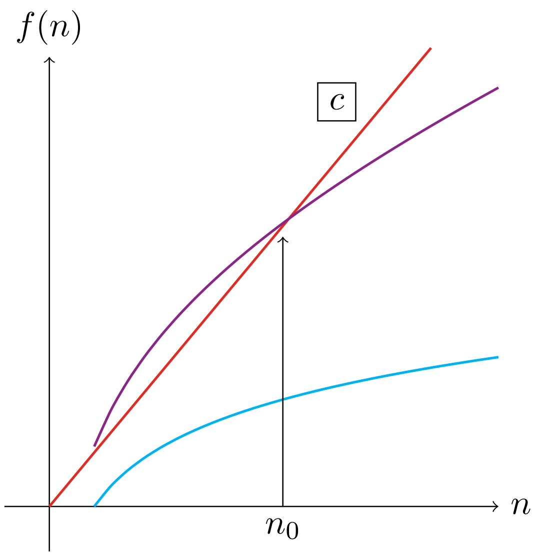
Big O notation – lower bound

Sometimes we also talk about a lower bound. This is the least amount of time that the algorithm will take.

Definition of Ω

A function $f(N)$ is $\Omega(g(N))$ if
for some constant c and
for values of N greater than some value n_0 then
$f(N) \geq cg(N)$

Bounds



Some common complexity names

1	constant time, independent of input
$logN$	logarithmic, only becomes slightly slower with increasing N
N	linear time, proportional to N
$NlogN$	log-linear or quasi-linear
N^m	polynomial, quadratic (m=2) and cubic (m=3), etc.
2^N	exponential
$N!$	factorial

Order of growth

The rate or order of growth of the running time is the most important part. We take the leading term of the equation and ignore coefficients, e.g., an algorithm with $T(n) = 3n^2 + 2n + 4$ steps will be described as having a worst case running time of $O(n^2)$, as will an algorithm with $T(n) = 5n^2 + 3$.

For you to do

Given the following running times, match them to their order of growth.

$f(n) = 4n^2$	$O(1)$
$f(n) = n^2 + 2n$	$O(\log n)$
$f(n) = 4$	$O(n)$
$f(n) = n^2(n-1)$	$O(n^2)$
$f(n) = n + 4n$	$O(n^3)$

Searching

- Check all the items : $\Theta(N)$ and therefore $O(N)$ and $\Omega(N)$
- Check through the items and stop when desired item found: $O(1)$ in the best case, $O(N)$ in the worst case
- You don't know what you are looking for exactly and have to return to the start with every single object to check: $O(N^2)$ but could get lucky with $O(1)$
- Generate all the permutations of the items and then go through them one by one selecting the item in position 1: $O(N!)$ but potentially $O(1)$ if you get lucky

Binary search

- Imagine you are searching for a number in a sorted list – what would be a good technique?
- Look at the midpoint, and if higher or lower, keep searching in that part of the list.

Lecture 8
Algorithms and complexity

COS10003 Computer Logic and Essentials (Hawthorn)

SWINBURNE UNIVERSITY OF TECHNOLOGY

Semester 1 2021

Recurrence

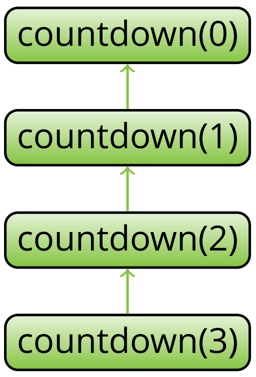
- Recurrence relations define consecutive elements in a sequence, where each element is defined by previous elements.
- We have seen some of these already, for example, the values in Pascal's Triangle: $\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$

Recursion

- The power of recurrence relations is being able to define recursive functions.
- Recursive functions are functions that are defined in terms of themselves.

Countdown!

```
function countdown(n):
   if n ≤ 0
      print "Blastoff!"
   else
      print n
      countdown(n-1)
   end if
   return
end
```



http://greenteapress.com/thinkpython2/html/thinkpython2006.html#sec62

For you to do

Finding the factorial of a number (assume n is not negative):

```
factorial(n):
    if         (base case)


    else


    end if
end
```

GCD again

Euclid’s algorithm used division/remainders instead of subtraction. This can be turned into a recursive version.

```
gcd(a, b):
    if (b == 0) then
       return a
    else
       return gcd(b, a mod b)
    end if
end
```

Complexity of recursive functions

This is outside the scope of this course, but for those who are interested, taking binary search as an example and counting steps:

$$T(n) = T\left(\frac{n}{2}\right) + O(1)$$

Time for the subproblem plus the time needed to set up and recombine

Complexity of recursive functions

This fits the pattern so that it can be solved by the Master Theorem:
$T(n) = aT\left(\frac{n}{b}\right) + f(n)$
There are three cases to the Master Theorem for determining the complexity of recursive functions, where the second is:
If $f(n)$ = $\Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \log n)$.
In this case, $a = 1$, $b = 2$; $n^{\log_2 1} = n^0 = 1 = f(n)$ which was constant time. So we get $\Theta(n^0 \log n) = \Theta(\log n)$.

How does recursion help us?

- Assist in proving properties about programs
- Helps us structure programs
- Also helps with designing data types and verifying their properties
- Allows the specification of infinite sets of values in a finite way

More maths

- A strongly related mathematical concept is induction.
- This is a proof technique that allows us to show that a property holds for all the natural numbers.
- This is one of the foundations for proving correctness of programs.

An example from earlier

We saw the equation $n + (n-1) + (n-2) + ... + 1$ in one of our algorithms. We can use induction to show that this is equal to $\sum_{i=1}^{n} i = n(n+1)/2$.
Base case: $n = 1$, $\sum_{1}^{i=1} i = 1 = 1 \times (1+1)/2$
Induction step: Assume the result is correct for $n = k$, $\underline{\sum_{i=1}^{k} i} = \underline{k(k+1)/2}$.
Now work with $\underline{k+1}$. We know that $\underline{\sum_{i=1}^{k+1} i} = \underline{\sum_{i=1}^{k} i} + \underline{(k+1)}$. The righthand side should be $\underline{(k+1)(k+2)/2}$.

An example from earlier

We saw the equation $n + (n-1) + (n-2) + ... + 1$ in one of our algorithms. We can use induction to show that this is equal to $\sum_{i=1}^{n} i = n(n+1)/2$.
Assuming that $\underline{\sum_{i=1}^{k} i} = \underline{k(k+1)/2}$, we get $\underline{\sum_{i=1}^{k+1} i} = \underline{k(k+1)/2} + \underline{(k+1)}$.
Multiply $(k+1)$ by 2 to get $\underline{k(k+1)/2 + 2 \times (k+1)/2}$, and refactor to get $\underline{(k+1)(k+2)/2}$. This is the result expected for $k+1$.

Another example

Show that the sum of odd numbers is equal to a square number, that is $\sum_{i=1}^{n}(2i-1) = n^2$.
Base case: $n = 1$, $\sum_{1}^{i=1} = (2 \times 1 - 1) = 1 = 1^2$
Induction step: Assume the result is correct for $n = k$, $\sum_{i=1}^{k}(2i-1) = k^2$.
Now work with $k+1$, and our expected result is $(k+1)^2$. We know that $\sum_{i=1}^{k+1}(2i-1) = (\sum_{i=1}^{k} 2i - 1) + (2(k+1) - 1)$, so assuming that $\sum_{i=1}^{k} 2i - 1 = k^2$, we get $\sum_{i=1}^{k+1} 2i - 1 = k^2 + 2(k+1) - 1$.
Expand to get $k^2 + 2k + 2 - 1 = k^2 + 2k + 1$. Factorise to find $(k+1)^2$. This is the result expected for $k+1$.

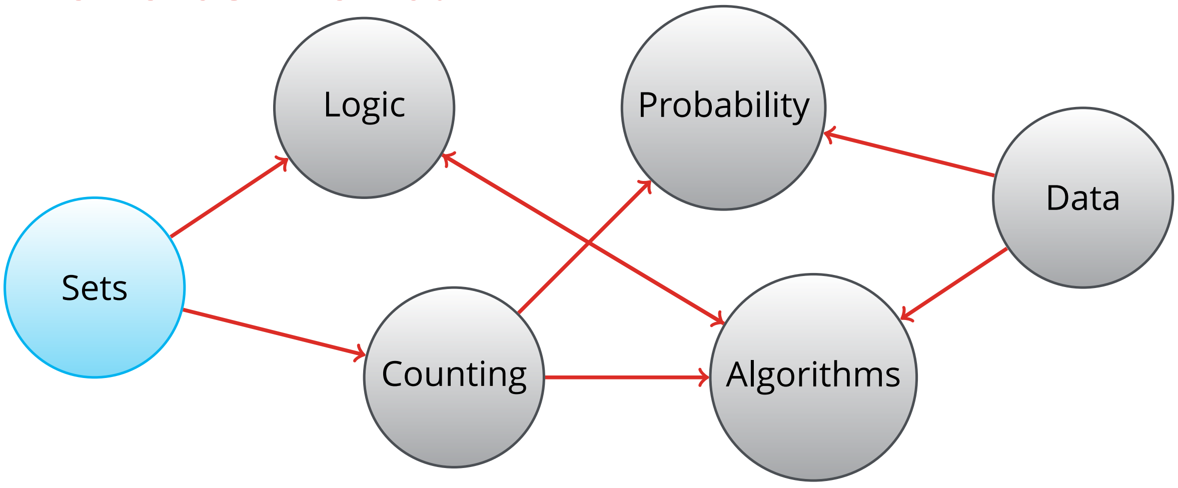
How is induction useful?

- Showing that the functions representing algorithms are in particular classes
- Proving program correctness, e.g., the correct execution of loops

Reflecting

- What are the two approaches to comparing two algorithms that perform the same function?
- What is the order of the different time complexities?
- How would you recognise a recursive function?

Where to next?



In which we explore basic graph theory.

Lecture 8
Algorithms and complexity

COS10003 Computer Logic and Essentials (Hawthorn)

SWINBURNE UNIVERSITY OF TECHNOLOGY

Semester 1 2021

Questions I still have

Topics I need to review