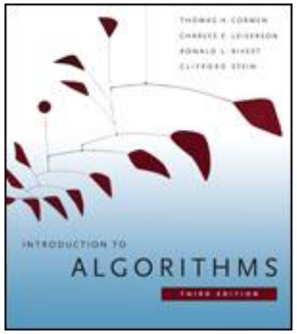
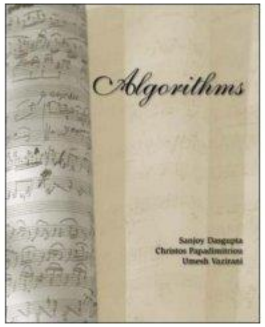
CO3002/7002 Analysis and Design of Algorithms

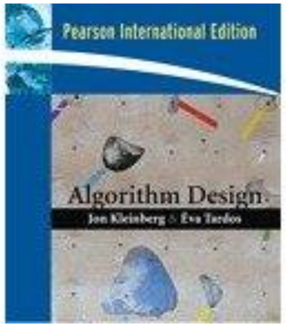
Stanley Fung
University of Leicester

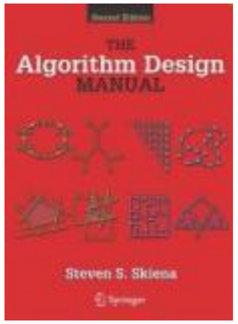
Module Administration

- Lecturer: Stanley Fung (pyf1@le.ac.uk)
- Website:
  - https://campus.cs.le.ac.uk/teaching/resources/CO3002
- Textbooks
  - Introduction to Algorithms [CLRS]
  - Algorithms [DPV]
    - Free download (draft) at http://www.cs.berkeley.edu/~vazirani/algorithms.html
  - Algorithm Design [KT]
  - Algorithm Design Manual [SSS]









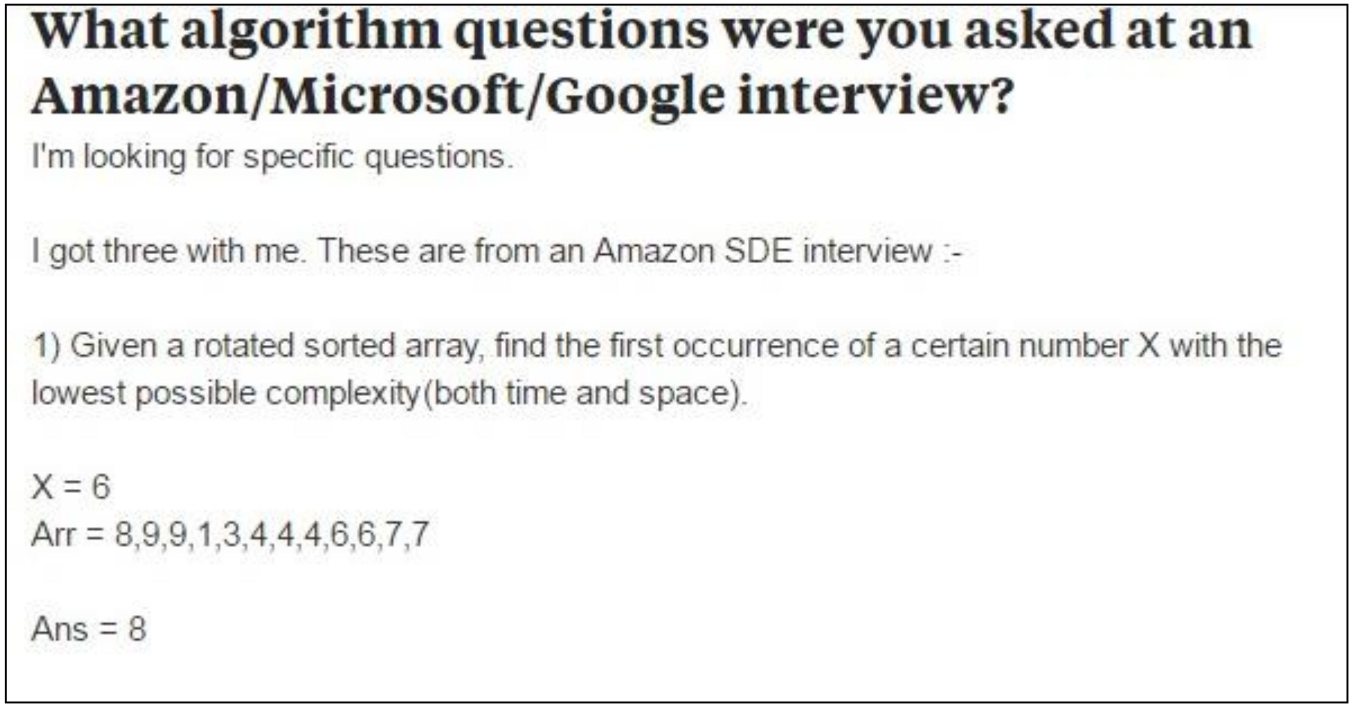
Coursework Assessments

- Assessment:
  - 40% coursework
  - 60% final exam
- Two types of coursework assessments:
  - Mini class tests x3 (15%)
    - ~20 mins each, routine “drilling” type exercises
    - Similar to (some of the) surgery exercises
  - Ongoing take-home coursework x2 (25%)
    - Require deeper understanding / a lot of thinking / design algorithms for unseen problems
- Exam will have a mix of different types of questions

Surgeries / Feedback Sessions

- ~~Thursday~~ 2-hour slot
- NEW: (almost) alternates between Thursday 4-6 BEN LT3 and Friday 3-5 BEN G85
  - Discuss (unassessed) exercises
  - Attempt exercises before class!
  - Other uses: class tests; Q&A for coursework; feedback and discussion of returned work
  - Usually takes < 1.5 hours

https://www.quora.com/What-algorithm-questions-were-you-asked-at-an-Amazon-Microsoft-Google-interview



What algorithm questions were you asked at an Amazon/Microsoft/Google interview?

I'm looking for specific questions.

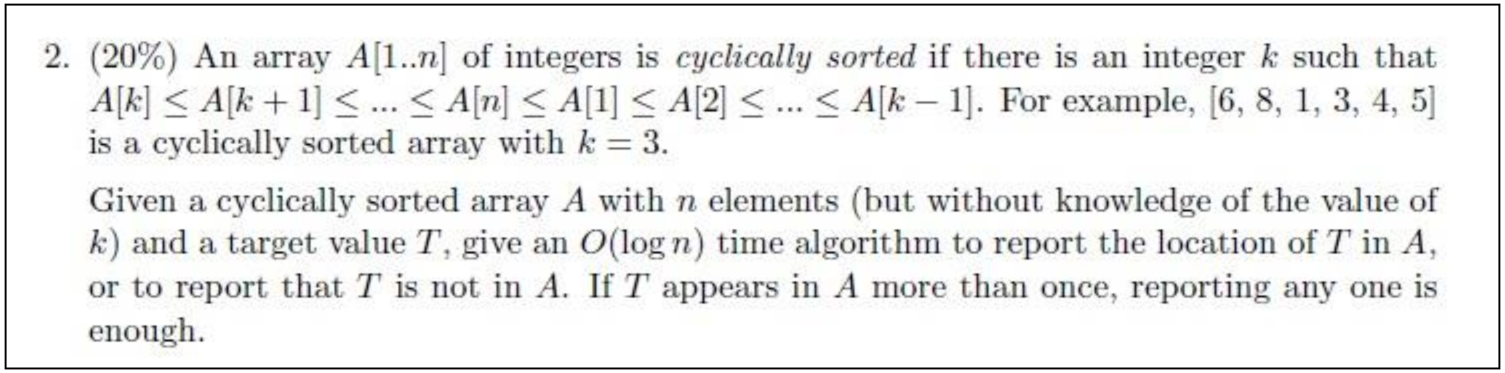
I got three with me. These are from an Amazon SDE interview :-

1) Given a rotated sorted array, find the first occurrence of a certain number X with the lowest possible complexity(both time and space).

X = 6
Arr = 8,9,9,1,3,4,4,4,6,6,7,7

Ans = 8

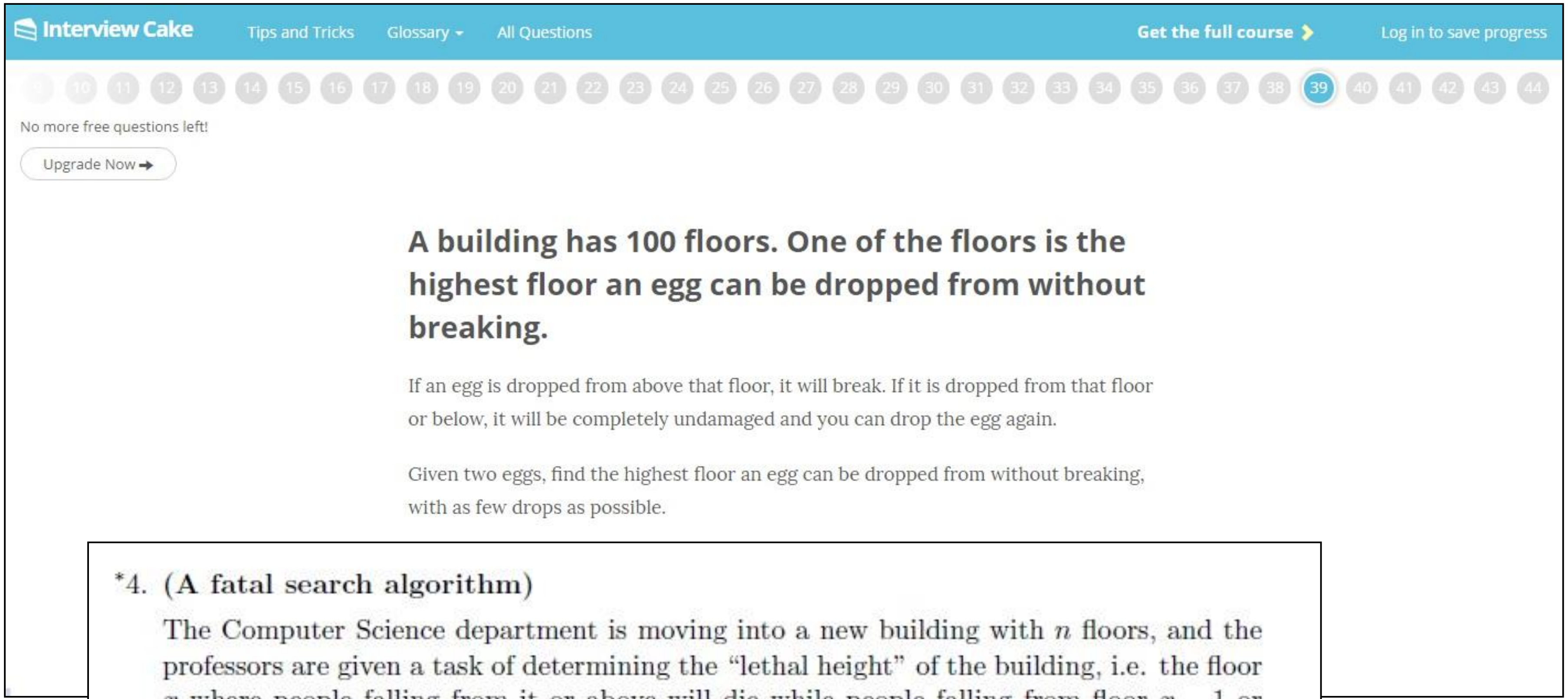
CO3002 2015/6 Assignment



2. (20%) An array $A[1..n]$ of integers is *cyclically sorted* if there is an integer k such that $A[k] \leq A[k+1] \leq ... \leq A[n] \leq A[1] \leq A[2] \leq ... \leq A[k-1]$. For example, [6, 8, 1, 3, 4, 5] is a cyclically sorted array with $k = 3$.

Given a cyclically sorted array A with n elements (but without knowledge of the value of k) and a target value T, give an $O(\log n)$ time algorithm to report the location of T in A, or to report that T is not in A. If T appears in A more than once, reporting any one is enough.

https://www.interviewcake.com/question/java/two-egg-problem

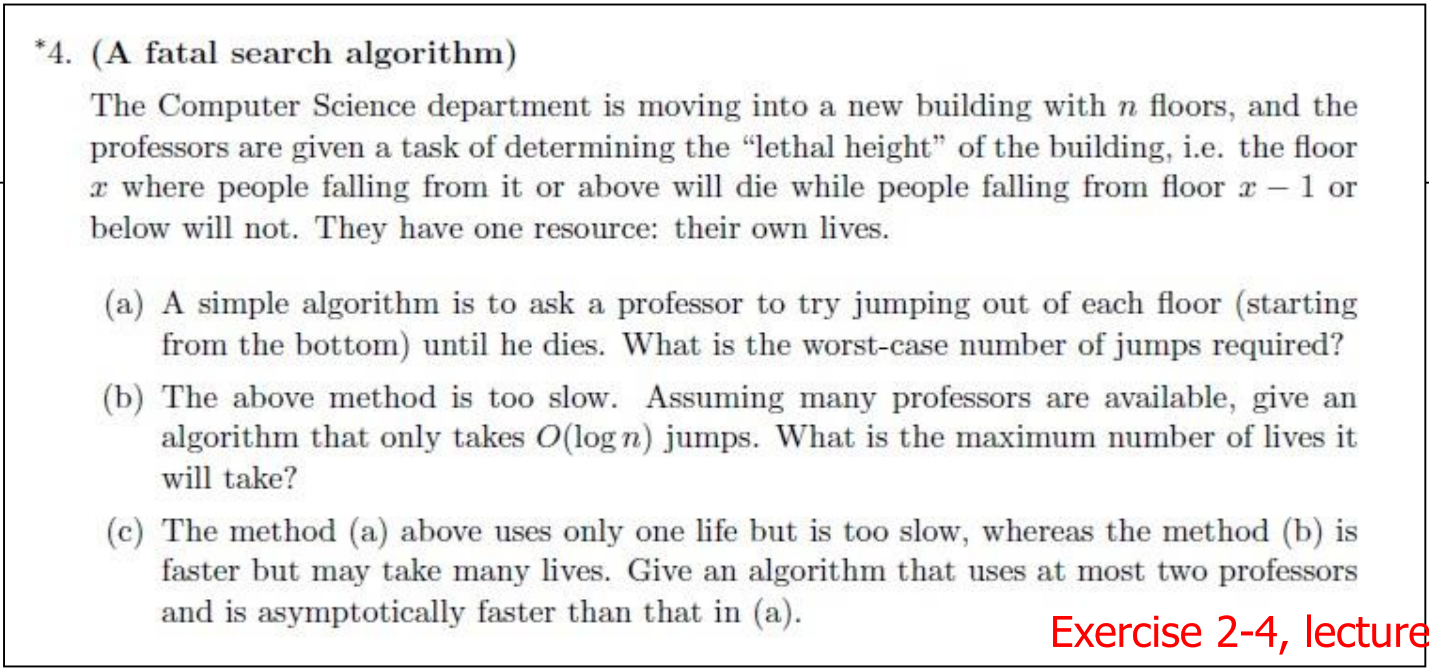


No more free questions left!

A building has 100 floors. One of the floors is the highest floor an egg can be dropped from without breaking.

If an egg is dropped from above that floor, it will break. If it is dropped from that floor or below, it will be completely undamaged and you can drop the egg again.

Given two eggs, find the highest floor an egg can be dropped from without breaking, with as few drops as possible.



*4. **(A fatal search algorithm)**

The Computer Science department is moving into a new building with n floors, and the professors are given a task of determining the "lethal height" of the building, i.e. the floor x where people falling from it or above will die while people falling from floor $x-1$ or below will not. They have one resource: their own lives.

(a) A simple algorithm is to ask a professor to try jumping out of each floor (starting from the bottom) until he dies. What is the worst-case number of jumps required?

(b) The above method is too slow. Assuming many professors are available, give an algorithm that only takes $O(\log n)$ jumps. What is the maximum number of lives it will take?

(c) The method (a) above uses only one life but is too slow, whereas the method (b) is faster but may take many lives. Give an algorithm that uses at most two professors and is asymptotically faster than that in (a).

Exercise 2-4, lecture notes

Module Contents

- Concepts of algorithms
- Algorithm analysis
  - Recurrence formula
- Algorithm design techniques
  - Divide and conquer
  - Greedy algorithms
  - Dynamic programming
- Fundamental problems
  - Sorting and searching
  - Graphs

Chapter 1
Basic Concepts

Designing the right algorithm for a given application is a major creative act… requires more than book knowledge. It requires a certain attitude – the right problem-solving approach. It is difficult to teach this mindset in a book…

- Steven Skiena, The Algorithm Design Manual

References:

[KT 2.1, 2.2, 2.4]

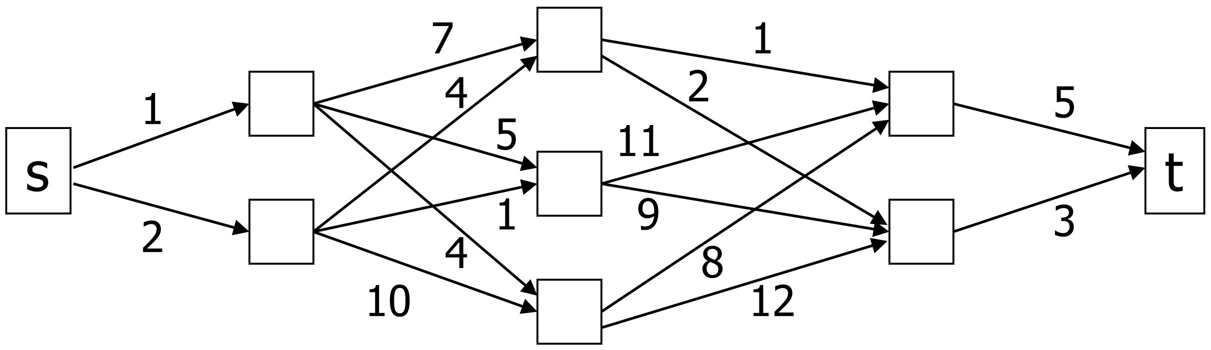
[CLRS 1, 2.2, 3]

[DPV 0]

[SSS 2]

Algorithms for Real-life Problems

- Many important real-life problems require efficient computational procedures:
  - Example 1: Find the common parts in these two DNA sequences?
    - AACGTCGTACCATGCGATGCATAAATCG
    - ACGTCAGTTCCATGAGAGCATTAATTCG
  - Example 2: What is the shortest path from s to t below?



Algorithms for Real-life Problems

- However, many seemingly simple problems have no efficient algorithms:
  - Example 3: Do some of the following numbers add up to exactly 1000?

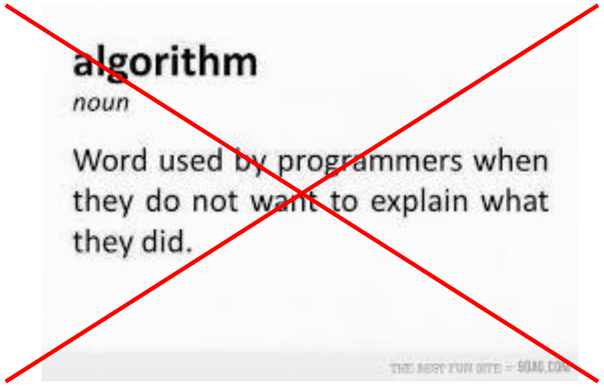
23	97	234	86	95	101	48	77	32	10
504	222	110	67	45	53	92	26	4	330

  - (Try “naïve” program at module webpage)
  - You can win a million dollars if you have an efficient algorithm (or a proof that there can’t be one)
- The study of algorithms:
  - Identify what problems can be solved efficiently
  - Learn techniques for designing efficient algorithms

Types of "Problems"

- What is a "problem"?
- *Decision problems*: only require a yes/no answer
  - Example: can a given map be coloured using at most 3 colours?
- *Optimization problems*: find the optimal (maximum/minimum) solution under constraints
  - Example: select from a set of conflicting activities that gives maximum total profit

What is an Algorithm?



~~**algorithm**~~

~~*noun*~~

~~Word used by programmers when they do not want to explain what they did.~~

- An *algorithm* is a step-by-step procedure for solving a computational problem
- A finite sequence of well-defined operations that halts in a finite amount of time
- The "idea" behind programs; what that remains unchanged when you translate the program into different languages

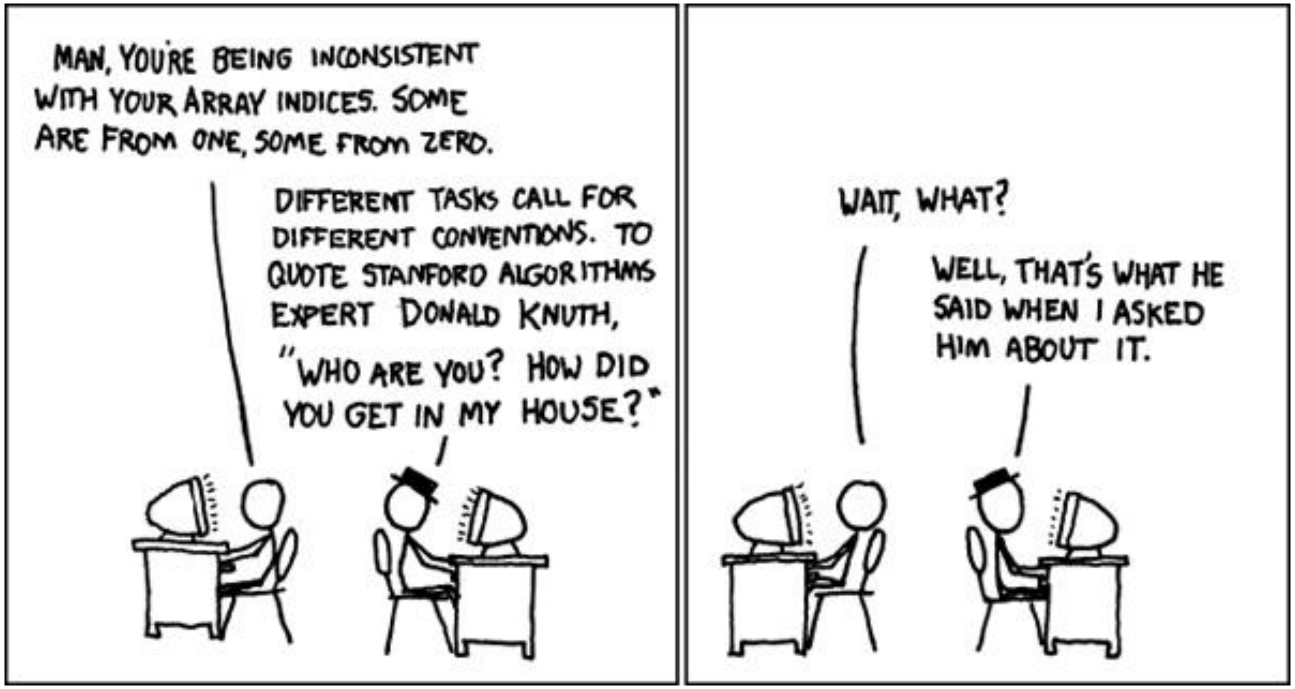
Algorithms ≠ Programs

- Programs are for computers to understand; algorithms are for humans
- Algorithms express the "ideas", are "high-level", and are less concerned with implementation details
- Algorithms are independent from programming languages and machine architectures
- But for precision, often expressed in *pseudocode*

```
/* input: array A[1..n] of size n
   output: a number x in A  */
x := A[1]
For i := 2 to n
  If (A[i] > x) x := A[i]
Output x
```

(what is it doing?)

(NB. in this module we sometimes assume array indices start at 0 and sometimes at 1, whichever is convenient. E.g. in the previous example the elements of A are A[1], A[2], …, A[n]. This is an example of implementation details that is irrelevant for algorithm analysis.)


MAN, YOU'RE BEING INCONSISTENT WITH YOUR ARRAY INDICES. SOME ARE FROM ONE, SOME FROM ZERO.
DIFFERENT TASKS CALL FOR DIFFERENT CONVENTIONS. TO QUOTE STANFORD ALGORITHMS EXPERT DONALD KNUTH, "WHO ARE YOU? HOW DID YOU GET IN MY HOUSE?"
WAIT, WHAT?
WELL, THAT'S WHAT HE SAID WHEN I ASKED HIM ABOUT IT.

https://xkcd.com/163/

Why Study Algorithm Design

- Straightforward algorithm is often not the best!
  - A deep understanding the structure of the problem leads to efficient algorithms
  - Often, these algorithms are non-trivial; it is not even clear why they perform the task correctly!
  - We study algorithms to learn these techniques
- Algorithm design ≠ programming tricks
  - We look for new ways of solving the problems
- "Algorithmic thinking": a different mindset
- A simple example (see surgery)
  - Given an array A[1..n] of numbers, compute all "contiguous sums", i.e. A[i]+A[i+1]+…+A[j] for all i and j (i<j)
  - How many additions in total?

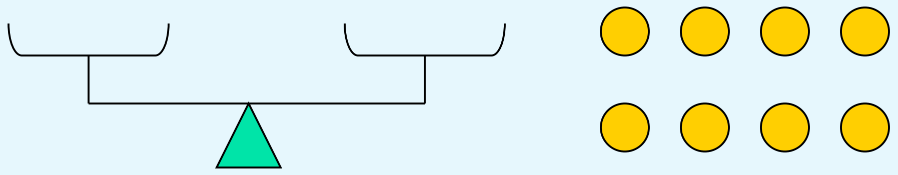
What Algorithms Can't Do

- Algorithms can't solve everything!
- *Halting problem*
  - Given a program P and an input I to the program P
  - Question: Is there an algorithm (a program) that reads P and I, and determine whether P will be in infinite loop when running with input I?
  - You may solve this problem by inspecting the source code, but *no algorithm can solve this!*
  - NOT that we have not yet discovered such an algorithm; we *proved* it is impossible to solve

Finding the Counterfeit Coin

- We use the following problem as a running example to illustrate the concepts

- Problem: Given 8 coins, one of which is counterfeit and is lighter. Using a pan balance, find the counterfeit coin
- Good algorithm?



Efficiency of Algorithms

- We analyse algorithms for their
  - Correctness
  - Efficiency
- Many algorithms for the same problem
  - Which is "the best"?
- Efficiency measures:
  - *Running time:* number of "steps" of the algorithm
    - "Step": a basic operation in a computer (that takes a constant amount of time)
  - Space (memory used)

Worst-case Analysis

- What is “the” running time? Different input has different running time!
  - E.g. searching for an element in an array of size n
  - Best case takes 1 step, worst case takes n steps
- *Worst-case running time:* the maximum number of steps the algorithm takes *over all possible inputs* (of a certain size)
  - This gives an upper limit of the running time for any input
  - Other measures, e.g. average-case running time, are possible, but
    - What is “average”? Need to assume knowledge of input distribution
    - Often difficult to analyse

Back to the Coin Example

- What is the efficiency measure?
  - "A step": using the pan balance once
  - Use as few steps as possible
- Consider the following algorithm A1:
  - Name the coins A, B, …, H.
  - Weigh A:B. If one side is lighter, report it and finish. Otherwise both are genuine.
  - Repeat for C:D, E:F, G:H.
- Analysis:
  - What is the best-case input? Worst-case input?
  - What is the worst-case running time of this algorithm?
  - Are there more efficient algorithms, in terms of the worst-case running time?

Upper Bound (of an Algorithm)

- The number of operations *sufficient* to solve a problem by an algorithm
  - The worst-case complexity of an algorithm
  - Good algorithms $\rightarrow$ small upper bounds
  - Upper bounds may be loose (not tight)
    - E.g. an algorithm is too complicated to be analysed exactly; we only give a "loose" proof that it takes at most n^3 steps, when in fact it never requires more than n^2 steps for any input
    - In contrast, an algorithm may take at most n^2 steps on *all* inputs, and it really takes n^2 steps on *some* inputs
    - In the latter case we sometimes (confusingly) refer this as the *lower bound of an algorithm*, i.e. the number of operations really taken by that algorithm on some input

Algorithm A1 Upper Bound

- 4, because it takes 4 steps in the worst case
  - (Even though if you are lucky it takes only 1 step)
- Is this bound tight?
  - Yes, since there is an input that this algorithm actually requires 4 steps (namely, coin G or H)
  - So the bound 4 is the best possible for this algorithm
  - If we want to improve, we need to find another algorithm
  - Not all algorithms are so easy to spot the worst case input

Upper Bound (of a Problem)

- Often, multiple algorithms for the same problem
- A problem has an upper bound x if there is an algorithm that solves it with x steps in the worst case

- Upper bound for coin weighing:
  - Any method to find the counterfeit coin using at most x weighings in the worst case
  - E.g. if there is another algorithm A2 for coin weighing that takes only 3 steps (in the worst case), then coin weighing now has an upper bound of 3
  - Can you find such an algorithm?
  - Can you find an even better one (2 steps)?

Lower Bound (of a Problem)

- The worst-case number of operations *necessary* to solve the problem by *any* algorithm
  - Not that we can't find a good algorithm yet; we prove that it is impossible
  - Larger lower bounds are better (in terms of analysis)
- How to prove lower bound? See Chapter 5

- Suppose we have an algorithm A2 that takes 3 steps, but cannot find one that takes 2 steps
  - Does NOT mean the lower bound is 2 (or 3)
  - When do we stop finding?
  - Can we prove it is impossible for any algorithm to do better than 3 steps? Or 2 steps?

Optimal Algorithms

- Upper bound = lower bound $\rightarrow$ *optimal algorithms*
- It is the quest of algorithm designers to design optimal algorithms
  - Match, or at least tighten the gap between, upper and lower bounds

- Suppose we know algorithm A2 takes 3 steps (upper bound)
- Suppose we also know any algorithm must use at least 2 steps (lower bound)
- There is a gap!
- Try to improve the upper bound to 2 (find better algorithm)?
- Or try to improve the lower bound to 3 (in which case A2 is optimal)

Growth of Functions

- Running time (*time complexity*) usually measured as function of input size (n)
  - We want to know how well an algorithm scales when input grows (*scalability*)
  - Not interested in small n, or a specific value of n
- Consider the running times of 3 algorithms on a machine that process 10^5 instructions/sec.

n	T1=10000n	T2=$500n^2$	T3=2^n
1	0.1s	0.005s	0.00002s
10	1s	0.5s	0.01024s
50	5s	12.5s	357 years
100	10s	50s	$4*10^{17}$ years

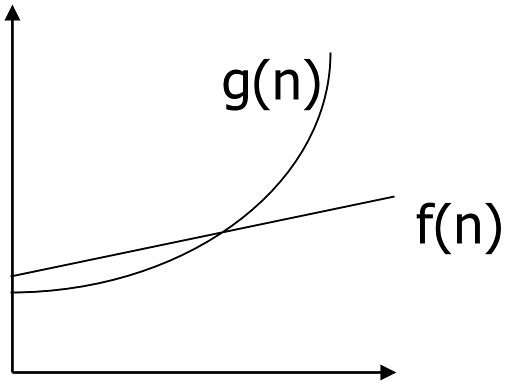
Growth of Functions

- Constants (e.g. 500) are not important
  - T1 increases linearly with n (i.e. n doubles, time doubles), whether it is 10,000 or 10
  - T2 increases quadratically with n (i.e. n doubles, time increases x4), whether it is 500 or 5
  - They depend more on the machine/program specifics, less on the “cleverness” of the algorithms
  - Getting a faster computer improves the constant, but will not improve the scalability
- We are interested in the *order of growth*, i.e. how quickly the function increases as n increases (*Asymptotic time complexity*)

The Big-O Notation

- The mathematical tool to capture the asymptotic notion
  - Formally, $f(n) = O(g(n))$ if $f(n) \leq c\, g(n)$ for all $n \geq n_0$, for some constants c, n_0
  - Informally: f(n) grows slower or same as g(n)
  - Hides lower order terms
    - e.g. if $f(n) = n^2 + 3n$, then n^2 "dominates" $3n$ when n is large, so we may as well ignore $3n$
  - Hides constant factors
  - Ignores small values of n

This use of the = sign is very common but informal and can be misleading; see notes



Common Running Times

- Some common time complexities (in increasing order):
  - $O(1)$: constant
  - $O(\log n)$: logarithmic
  - $O(n)$: linear
  - $O(n \log n)$
  - $O(n^2)$: quadratic
  - $O(n^3)$
  - $O(2^n)$: exponential
- Can also be used with multiple variables
  - E.g. two input arrays of sizes m and n; graph with n vertices and m edges
  - $O(m+n)$ (linear), $O(mn)$, etc.

Tractability

- We usually say the running time is *efficient* if it is a polynomial in the input size, i.e. $O(n^k)$ for constant k
- Otherwise (e.g. exponential, $O(2^n)$), we regard it as inefficient
  - Exponential functions grow much faster
  - In practice, it is very rare to see running times of n^{100} or 1.001^n
- Problems that do not have polynomial time algorithms are called *intractable*
  - Usually, nothing significantly better than brute force (search all possibilities) is known

Relatives of Big-O

- Big-Omega:
  - $f(n) = \Omega(g(n))$ if $f(n) \geq c\, g(n)$ for all $n \geq n_0$, for some constants c, n_0
  - Informally: f(n) grows faster or same as g(n)
  - Usually used for lower bounds
- Big-Theta:
  - $f(n) = \Theta(g(n))$ if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$
  - Optimal/tight bounds
- Rough analogy:
  - $O \leftrightarrow \leq$
  - $\Omega \leftrightarrow \geq$
  - $\Theta \leftrightarrow =$

Examples of Big-O,Ω,Θ

- All these are correct:
  - $2n = O(n)$, $0.25n^2 - n - 1 = O(n^2)$, $4n = O(n^3)$
  - $2n = \Omega(n)$, $1.5\ n \log n = \Omega(n)$
  - $2n = \Theta(n)$, $2n^3 - 6n^2 + 7n - 8 = \Theta(n^3)$
- All these are not correct:
  - $n^2 = O(n \log n)$
  - $3n^2 = \Omega(2^n)$
  - $n = \Theta(n^2)$

Simple Complexity Analysis

- Given an algorithm (pseudocode), how to analyse its complexity?
- "Standard" primitive operations take constant time each
  - Arithmetic (+,-,x,/), comparison (=,>,<), read/write memory, …
- Consecutive statements
  - Complexity of {S1; S2;}

  = complexity of S1 + complexity of S2

  = max(complexity of S1, complexity of S2)

Simple Complexity Analysis

- Loops:
  - Complexity = complexity of statements inside the loop * number of times the loop is executed
- If-then-else:
  - Complexity of if (A) then B else C = max(complexity of A+B, complexity of A+C)
- Function calls:
  - Complexity of function calls should be analysed separately from the calling routine

Example

- What is the time complexity of this algorithm?
  - What is it doing?

```
/* input: 2-D array A[1..n][1..n]
   output: a number max */
Max := 0
For i := 1 to n
  sum := 0
  For j := 1 to n
    sum := sum + A[i][j]
  If (sum > max) max := sum
Output max
```

- (NB: mistake in lecture notes…)