```racket
;; An Introduction to Algorithms: Shape, Space and Time

;; A talk given twice at Villiers Park on 26/11/2015, to two very bright groups of teenagers.

;; In both cases we got through to the memoization of fib, one time in just over one hour, one time in
an hour and a half.

;; Written for PLT DrRacket, version 6.1, and this time using the racket language, just so we can
memoize later on

;; Again using the dojo format, where two people from the audience do all the typing, the audience
does the programming
;; and I try to limit myself to asking leading questions.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; Firstly, I want to say that this talk is full of lies.
;; Adults have been lying to you all your life, and it's past time that you learned to spot when
they're lying.
;; So I'm going to be very impressed and pleased if you can catch me out in any of my lies.
;; That's a skill that's worth having if you remember nothing else from this talk.

;; Zebras are Blue

;; Isn't anyone going to pull me up on that?
;; I mean, I've just told you that I'm going to lie to you, and
;; I've just told you that I want you to catch me out when you think I'm lying, and
;; I've just told you a very obvious lie.

;; Surely someone is brave enough call me on it?
;; You don't have to be rude about it, or even particularly accusatory.
;; You can put your hand up politely, and you can say :
;; "Excuse me, I'm probably wrong here, but I think I remember reading somewhere that some zebras
;; are black and white. Is that a myth?".

;; OK, Zebras are a kind of space fish, which is blue, and they live on the Moon.

;; Good. At least one person here is brave enough to tell the Emperor when he's not wearing any
;; clothes, at least once the Emperor has said it's ok. But anyway, I'm going to stick in lots more
;; lies in this talk, but they won't be as obvious. I'm going to see how many I can sneak past you.

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; We've written some programs in Scheme.

;; A program is a recipe for a process, and today I want to talk about the shapes of the processes
;; that programs create as they run.

;; I'm going to give you some little programs, and you're going to play the role of
;; the computer and execute them.

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

#lang racket

;; First off, I want to talk about 'the substitution model', which is a way of understanding what a
;; computation is. You could build a computer to work this way if you liked. I have done.

;; So the other day I showed you how to make a function to square things
(lambda (x) (* x x))
;; And I told you how to name it
'(define square (lambda (x) (* x x)))
;; But we can do both things at once:
'(define (square x) (* x x))
;; Now this is called 'syntactic sugar'. And what happens is that the evaluator, when it sees an
;; expression where the thing it's supposed to name is a list like (square x)
;; quietly rewrites it to be (define square (lambda (x) (* x x) ).
```

```scheme
;; A very famous man once said that "syntactic sugar leads to cancer of the semicolon".
;; And there's very little in Scheme, but we do have this one bit,
;; because Alonzo Church himself wouldn't have used lambdas if he'd had to write out l-a-m-b-d-a all
;; the time, and this bit of sugar makes programs a little easier to read and write at the cost of
;; hiding what's going on.

;; Today, I think it will make it easier to reason about what we're going to reason about if we
;; use this shorthand

;; So suppose we've got this function
(define (square x) (* x x))
;; And this function
(define (pythag x y) (+ (square x) (square y)))

;; And we type
(pythag 3 4)
;; into the evaluator, what happens?

;; Well, in a substitution model computer, this:
(pythag 3 4)
(+ (square 3) (square 4))
(+ (* 3 3) (square 4))
(+ 9 (square 4))
(+ 9 (* 4 4))
(+ 9 16)
(+ 9 16)
25
;; And let's call that seven steps of evaluation

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; First of all, let's think about factorials.

;; How many ways are there to arrange 3 things?

;; abc acb bac bca cba cab. So six. Are we done?

;; Let's get rid of c

;; ab ab ba ba ba ab

;; Every way of arranging three things is a way of arranging two things, with c put in somewhere

;; ab -> cab, acb, abc
;; ba -> cba, bca, bac

;; So the number of ways you can arrange 3 things is 3 times the number of ways you can arrange 2
things.
;; And the number of ways you can arrange 2 things is 2 times the number of ways you can arrange 1
thing.
;; And the number of ways you can arrange 1 thing is 1.
;; ( Which is 1 times the number of ways you can arrange no things, which is also 1 )

;; This number is called the factorial, and it comes up so often that it has its own bit of notation.
;; We usually write 3! for 3 factorial.

;; A mathematician might write:
;; 0! = 1
;; n! = n * (n-1)!

;; We can take this mathematical definition and turn it directly into a program

(define (factorial n)
  (if (= n 0) 1
      (* n (factorial (- n 1)))))

(factorial 3)
```

```
;; Now suppose you're the computer, how do you execute that program?

(factorial 3)
(if (= 3 0) 1 (* 3 (factorial (- 3 1))))
(if #f 1 (* 3 (factorial (- 3 1))))
(* 3 (factorial (- 3 1)))
(* 3 (factorial 2))
(* 3 (if (= 2 0) 1 (* 2 (factorial (- 2 1)))))
(* 3 (if #f 1 (* 2 (factorial (- 2 1)))))
(* 3 (* 2 (factorial (- 2 1))))
(* 3 (* 2 (factorial 1)))
(* 3 (* 2 1))
(* 3 2)
6


;; 11 steps

;; Now let's get rid of the book-keeping in the middle, where we do the function call and the
;; subtraction and the if, which is the same every time, and just keep the interesting bits.

(factorial 3)
(* 3 (factorial 2))
(* 3 (* 2 (factorial 1)))
(* 3 (* 2 1))
(* 3 2)
6

;; 5 steps, because we're counting steps differently now

;; Notice how the program grows, and then when you get to (factorial 1) it bounces and starts to
;; shrink until we're back to 1 number, which is the answer.

;; It's like it spends the first half of its time planning a calculation,
;; and the second half working it out.

;; Suppose we wanted (factorial 30), what would that look like?

;; (factorial 30)
;; ....
;; ....
;; (* 30 (* 29 (* 28 (* 27 (* .......                           (* 3 (factorial 2)))))))
;; (* 30 (* 29 (* 28 (* 27 (* .......                           (* 3 (* 2 (factorial 1))))))))
;; (* 30 (* 29 (* 28 (* 27 (* .......                           (* 3 (* 2 1)))))))
;; (* 30 (* 29 (* 28 (* 27 (* .......                           (* 3 2))))))
;; ....
;; ....
;; (* 30 8841761993739701954543616000000)
;; 265252859812191058636308480000000


;; It takes 30 steps to build up the sum, and 30 steps to collapse it down,
;; and the longest line, in the middle, is 30 numbers long.

;; What about (factorial 300)?

;; A big number, that, but it only takes 100 times longer to compute it than it did (factorial 3),
;; and in the middle, we need to store a big long string of computations waiting to be done, but
;; it's only 100 times longer than the string of stuff for (factorial 3)

;; We call this a linear recursion.

;; Here's the shape of a  linear recursion. It's a triangle in space and time.

;-
;---
```

```
;-----
;-------
;-----
;---
;-


;; Generally, we ignore the details about whether it's n steps, or 7*n steps, because how many steps
;; it is depends on how you count them, and how long each step takes depends what sort of computer
;; the program is running on.

;; So we tend to think that if the time taken is 15*n, then the 15 is sort of a fiddly detail and
;; the n is the important bit!

;; Normally that's ok, because the difference between 15 milliseconds and 1 millisecond is not
;; important, and neither is the difference between a million years and 15 million years.

;; We say that we need O(n) time, and O(n) space.

;; Occasionally, you'll fall into the annoying zone where it's the difference between 1 day and two
weeks.
;; At that point you do need to start caring about constant factors. But that's surprisingly rare.

;; At any rate, with an O(n) algorithm, our problem is surprisingly tractable, and we can find quite
large factorials.
(factorial 1000)

;; Notice that if we stop our process in the middle, we'll need to write down quite a lot of stuff to
be able to restart it from the point where it left off.

;; We can draw what's called a call graph of the process, which shows us how the data flows
'((factorial 7) --> (factorial 6) --> (factorial 5) --> (factorial 4) --> (factorial 3) --> (factorial
2) --> (factorial 1) --> (factorial 0))
'( 5040        <-- 720          <-- 120           <-- 24            <-- 6             <--
2             <--  1           <--  1          )

;; [ factorial.svg ]

(display "=========================================================================================\n")

;; Now, if we look in the middle of the factorial process, we'll find that

(factorial 7)
;; is equal to:
(* 7 6 5 4 3 2 1)

;; So taking that as inspiration, here's a different way of computing the same function:

(define (fact n) (fact-iter n 1))

(define (fact-iter n total)
  (if (= n 0) total
      (fact-iter (- n 1) (* n total))))

(fact 3)

;; We get the same answers, but what does the computation look like now?

(fact 3)
(fact-iter 3 1)
(fact-iter 2 3)
(fact-iter 1 6)
(fact-iter 0 6)
6

;; What would (fact 30) look like now?
```

```scheme
(fact 30)
(fact-iter 30 1)
(fact-iter 29 30)
(fact-iter 28 870)
(fact-iter 27 24360)
(fact-iter 26 657720)
;;....
;;....
(fact-iter 0 265252859812191058636308480000000)
265252859812191058636308480000000

;; We're no longer using up a lot of space storing the computations we intend to do.
;; At every step, we need only store two numbers, our counter, and our total.
;; So it will be much easier to stop and start our computation in the middle, we only have to write
down e.g.
(fact-iter 26 657720)
;; and that will carry on where we left off

;; This pattern is called a linear iteration, and it's still O(n) in time, but it's O(1) in space.

;; With an algorithm like this, it may take a long time to do the computation, but we don't really
;; have to worry about running out of memory.

;; With the first process, we might set off a calculation that we expect to take hours, and come back
;; later to find that the whole thing had failed because the computer had run out of space to store
;; the computation.

;; Notice that in both cases, the programs are 'recursive' in the sense that the procedures call
themselves.

;; What makes the difference is the shape that the programs make as they run.


;; Here's the shape of a linear iteration
;--
;--
;--
;--
;--
;--
;--


;; In a lot of languages, iteration has a privileged place, and it gets its own iteration construct,
;; which might be a for/next loop, or loop/exit/end, or do/while, or while/do.

;; In some languages, that's the only sort of program you can write!

;; Notice that we had to do some work to turn our linear recursion into a linear iteration, and it's
;; not quite so obvious that the second program is correct and bug free.

;; There's a general pattern here. Some ideas are more natural to express recursively, and some are
;; more natural to express iteratively.

;; If your language forces you to use iterations only, then it forces you to do some of the work
;; that the computer should do for you.

;; But often, if you can express your idea as an iteration, you'll generate a nicer process.




;; We can also draw what's called a call graph of the iterative process, which shows us how the data
flows
'((fact 7) --> (fact-iter 7 1) --> (fact-iter 6 7) --> (fact-iter 5 42) --> (fact-iter 4 210) -->
(fact-iter 3 840) --> (fact-iter 2 1680) --> (fact-iter 1 3360) --> (fact-iter 0 3360) --> 3360)
```

```scheme
;; [ fact-iter.svg ]

;; Notice that the shape is kind of different and kind of the same.

;; The data flows along the chain, but at the end it falls off the end, it doesn't come back, and
;; no more work is done on it once it reaches the end of the chain.

(display "=========================================================================================\n")

;; Now let's look at what is thought to be the worst example in computer science:

;; The Fibonacci Numbers are defined recursively, and they're not important enough to have their
;; own symbol, like the factorials do, although there are plenty of cranks who will tell you that
;; they are everywhere in Nature whooooo.

;; If you ever hear anyone talking about the mysterious omnipresent wonder of the fibonacci
;; numbers, run.

;; However like all really simple ideas, they do come up from time to time, in for instance
;; poetry, binary strings, idealized bees, rabbit counting, flowers, algorithms, data structures and
so on.

;; A mathematician might define the fibonacci numbers thus:

;; FIB(0) = 1
;; FIB(1) = 1
;; FIB(n) = FIB(n-1) + FIB(n-2)

;; And we, in our innocence, might then translate this recursive definition into this recursive
function:

(define (fib n)
  (if (< n 2) 1
      (+ (fib (- n 1))
         (fib (- n 2)))))

(fib 1)
1
;; 1 step

(fib 1)
1
;; 1 step

(fib 2)
(+ (fib 1) (fib 0))
(+ 1 (fib 0))
(+ 1 1)
2
;; 4 steps  ( 1 + 2 + 1 )


(fib 3)
(+ (fib 2) (fib 1))
(+ (+ (fib 1) (fib 0)) (fib 1))
(+ (+ 1 (fib 0)) (fib 1))
(+ (+ 1 1) (fib 1))
(+ 2 (fib 1))
(+ 2 1)
3
;; 7 steps ( 2 + 3 + 2 )

(fib 4)
(+ (fib 3) (fib 2))
(+ (+ (fib 2) (fib 1)) (fib 2))
(+ (+ (+ (fib 1) (fib 0)) (fib 1)) (fib 2))
(+ (+ (+ 1 (fib 0)) (fib 1)) (fib 2))
(+ (+ (+ 1 1) (fib 1)) (fib 2))
(+ (+ (+ 1 1) 1) (fib 2))
(+ (+ 2 1) (fib 2))
(+ 3 (fib 2))
(+ 3 (+ (fib 1) (fib 0)))
(+ 3 (+ 1 (fib 0)))
(+ 3 (+ 1 1))
(+ 3 2)
5
;; 13 steps ( 4 + 5 + 4)

(map fib (range 30))

;; You might notice that the larger fibs take a while to compute.

;; Why is that?

;; Let's see if we can draw the call graph.

;; [Draw the call graph]
;; [ show that the fringe has (fib n) elements, and that there are (- (fib n) 1) interior nodes]
;; [Notice that the every interior node represents a split step and a combine step, and
;;       every fringe node represents an evaluate-to-one step]

;; [insert fib.svg]

;; This is called a tree recursion, and it is exponential in time, and linear in space.

;; Now this example is pretty. I like it very much, which is why I have put it third on
;; my list of 'classic algorithms'

;; Why do I call it 'proverbially the worst example in computer science?'

;; It's because it teaches you four lessons, and they're all wrong!

;; Lesson I : Fibonacci Numbers are hard to compute...

;; Here I am computing them by hand
1
1
(+ 1 1) ;-> 2
(+ 2 1) ;-> 3
(+ 3 2) ;-> 5
(+ 5 3) ;-> 8
(+ 8 5) ;-> 13
(+ 13 8) ;-> 21

;; .. and so on and so forth. Now I am not at all sure that I am going to be able to get to (fib 40)
;; before the computer can, but I am damned sure that I am going to get to (fib 60) before it does.

;; Think about that for a minute.

;; I am going to take 60 additions to get to (fib 60)
;; The computer is going to take two and a half TRILLION steps to get there.

;; Now if you can calculate something faster than your computer can, you are doing something very
wrong.
;; And if fact I can write an iteration that calculates them just like I was doing above.

;; It looks like this:

(define (fib-iter a b n)
  (if (> n 0)
      (fib-iter b (+ a b) (- n 1))
      a))

(define (ifib n) (fib-iter 1 1 n))
```

```
(ifib 4)
(fib-iter 1 1 4)
(fib-iter 1 2 3)
(fib-iter 2 3 2)
(fib-iter 3 5 1)
(fib-iter 5 8 0)
5

;; Ten Thousandth Fibonacci Number anyone?
(ifib 10000)

;; Take a moment to reflect that this enormous number (times 3 minus 2) is the number of steps that
the tree-recursion
;; algorithm above would take to calculate (fib 10000)

;; Lession II

;; Recursion is BAD!

;; Some algorithms are naturally recursive, writing them as iterations can always be done, but it is
often a pain.
;; One nice example is the merge sort algorithm

;; A child couldn't sleep, so her mother told a story about a little frog,
;;  who couldn't sleep, so the frog's mother told a story about a little bear,
;;     who couldn't sleep, so bear's mother told a story about a little weasel
;;       ...who fell asleep.
;;     ...and the little bear fell asleep;
;;  ...and the little frog fell asleep;
;; ...and the child fell asleep.

;; Also see Lesson III

;; Lesson III

;; Tree Recursion is BAD!

;; Trees are everywhere.
;; Obviously, there are actual trees (the larch, the oak,...)
;; But your body is a tree of bones
;; And the internet is a tree of links
;; And your computer network is a tree of wires
;; And a recipe is a tree of cookery tasks
;; And a to-do list is a tree of things to do
;; And computer programs are trees of functions.
;; This is really obvious in Scheme, but there aren't many computer languages where it isn't just as
true.
;; And even fewer that you'd actually use for any purpose. The only exception I can think of is
programming directly in machine code.
;; And even then, you wouldn't do that. You'd use assembler languages. And even assemblers build
little trees as they run.

;; If you try to write programs about trees without using tree recursion, well, it can be done. But it
is very hard.
;; And if you try to write programs not about trees, that doesn't leave a huge number of programs.
Most of which have already been written.

;; Lesson IV

;; Tree Recursion is a BAD way of computing the Fibonacci Numbers!
;; OK, I will admit that there is sort of a case for saying that this is true.

;; But actually it's not terrible. The problem with the program above is not actually the fact that
each function call results in needing to make two more function calls.

;; It is that the program is doing the same thing over and over and over again.
```

```scheme
;; There are two solutions to this:

;; One is called dynamic programming, where you look very carefully at what information you need to
calculate and in what order and you arrange your computation
;; In the fibonacci case, dynamic programming is very easy to do, and I have used it above to turn the
fib function into the ifib function, which is an iteration.
;; Dynamic Programming is an ancient and honourable technique of algorithm design, but it often
requires careful thought and intricate design.
;; Which is all very excellent and satisfying, until it breaks and you have to start looking for bugs.

;; And the other way is called memoization, which is when you get the function itself to remember the
computations it has already done,
;; and if you ask the function to compute something it has already computed, then it just looks it up,
instead of recalculating.

;; Memoization is one of those 'classic patterns' that you can use over and over again.

;; Here is the function that memoizes things. It's not very complex, even though it's got some lambdas
in it.
(define (memoize f)
  (let ((table (make-hash)))
    (lambda args
      (dict-ref! table args (lambda () (apply f args))))))
;; It takes a function, and it gives back another function that remembers when it's done something
before.

;; And here's mfib, which is the tree recursion from earlier, with the extra word memoize in it.
(define mfib
   (memoize (lambda (n)
      (if (< n 1) 1
        (+ (mfib (- n 1)) (mfib (- n 2)))))))

;; Here's that ten-thousandth fibonacci number again:
(mfib 10000)



(display "=========================================================================================\n")
;; OK, there is no way that we are going to have got this far in an hour, but here are some more
algorithms that you might want to think about:



;; Whether he's trying to work out the largest type of square carpet
;; tiles he can get away with when covering a complicated floor, or
;; simply needs to reduce a vulgar fraction to its lowest form, There
;; comes a time in the life of every man when he must find the
;; greatest common divisor of two large integers.

;; Euclid shows us a better way

;; Life is the art of deriving great general principles from the
;; careful study of one well-understood simple example. At least that's
;; the way it works in mathematics.

;; Euclid's algorithm (how many steps does (gcd (fib 30) (fib 31)) take?)



(define (gcd a b)
  (printf "gcd(~a,~a)\n" a b)
  (cond ((= a 0) b)
        ((= b 0) a )
        ((< a b) (gcd a (remainder b a)))
        (else    (gcd (remainder a b) b))))
```

```scheme
(gcd (fib 30) (fib 31))

(display "=========================================================================================\n")

;; Ackerman's function (what's the largest pair x,y where (A x y) and the process that computes it are
even remotely comprehensible to the human mind?

(define (A x y)
  (cond ((= y 0) 0)
        ((= x 0) (* 2 y))
        ((= y 1) 2)
        (else (A (- x 1) (A x (- y 1))))))

(display "=========================================================================================\n")

;; The Strict Egyptian Fractions Algorithm

;; For reasons unknown, and doubtless completely weird, the Ancient Egyptians, although happy to
manipulate fractions,
;; didn't like fractions which didn't have one on the top, and didn't like to repeat themselves either:

;; The sanest argument I've ever heard for why this might have been involves pizzas

;; Imagine 12 people are trying to share 13 pizzas

;; Everyone should end up with 13/12 pizzas, which is not enormously easy to arrange, but suppose
instead of 13/12 you had a rule that you had to split
;; fractions up into distinct unit fractions, e.g.:

;; 13/12 = 1/2 + 1/3 + 1/4

;; You'd easily be able to see that you could make 12 half-pizzas, and 12 1/3 pizzas, and 12 1/4
pizzas, which is easy.

;; As far as I know, there isn't any archaeological evidence for pizza in ancient Egypt, but some
might turn up any day.

;; Anyway the Strict Egyptian Fraction algorithm goes like this (notice that 1/n = 1/(n+1) + 1/n(n+1)
):

3/2 ;; <- bad, it's got a 3 on the top
(+ 1/2 1/2 1/2) ;; still bad, three things the same

;; notice that 1/2 = 1/3 + 1/6

(+ 1/2 1/2 1/3 1/6) ;; better, only two things the same, do that again!

(+ 1/2 1/3 1/3 1/6 1/6)  ;; arrgh, it's getting worse

;; notice that 1/3 = 1/4 + 1/12

(+ 1/2 1/3 1/4 1/6 1/6 1/12 ) ;; looking good, but still too many 1/6s

;; notice that 1/6 = 1/7  + 1/42

(+ 1/2 1/3 1/4 1/6 1/7 1/12 1/42) ;; done.

;; That is how the Ancient Egyptians would have expressed the idea of three halves,
;; which probably explains why I don't know the names of any ancient Egyptian mathematicians.

;; Now, the Strict Egyptian Fractions Algorithm, is, as far as I know, only interesting in the sense
of "it sometimes takes an enormously long time to stop".

;; And so my final question for you is:

;; Will it always finish no matter what number you start it with, or is there a case where it just
goes on forever,
;; getting worse and worse and worse?
```