```racket
#lang racket

(define (contains-h l n comp)
  (cond
    [(empty? l) #f]
    [(comp (car l) n) #t]
    [else (contains-h (cdr l) n comp)]))

(define (contains-1d l n)
  (contains-h l n =))

(define (contains-2d l n)
  (contains-h l n contains-1d))

(define (a-list a b comp-ls)
  (cond
    [(< b 2) (list)]
    [else (begin
            (define n (expt a b))
            (cond
              [(contains-2d comp-ls n) (a-list a (- b 1) comp-ls)]
              [else (cons n (a-list a (- b 1) comp-ls))]))]))

(define (length-2d l)
  (if (empty? l)
      0
      (+ (length (car l)) (length-2d (cdr l)))))

(define (solve-h a accum b-max)
  (cond
    [(< a 2) (length-2d accum)]
    [else (solve-h (- a 1)
                   (cons
                    (a-list a b-max accum)
                    accum)
                   b-max)]))

(define (solve max)
  (solve-h max (list) max))

(solve 100)
```