As a first problem we shall write the algorithm to add two n digit numbers:

Purpose - Illustrate how to formulate recursive solutions to problems

1. Take an example:

1	0	1		→ Adding Three single digit numbers
9	6	4	9	
0	6	3	5	
1	0	2	8	4

→ We shall assume a primitive called add_single which can add Three single digit numbers.

2. Figure out the solution of the given input in terms of solution of a part of the input:

add (9649, 0635) = 9649 + 0635 (↳ not allowed)

= 9640 + 0630 + 9 + 5

= 9640 + 0630 + 10 + 4

= (964 + 063 + 1) * 10 + 4

reduced to addition over smaller numbers. But there is a carry

add (9649, 0635) = addc (9649, 0635, 0)

(↳ add with carry)

= 9640 + 0630 + 9 + 5 + 0

(↳ add_single_carry)

= 9640 + 0630 + 14

= 9640 + 0630 + 10 + 4

= (964 + 063 + 1) * 10 + 4

= addc (964, 063, 1) * 10 + 4

= convert (addc (964, 063, 1), 4)

addc (x, y, c) = convert (addc (qx10, qy10, qsum10), rsum10)

where qx10 = quotient (x, 10)
qy10 = quotient (y, 10)
rx10 = remainder (x, 10)
ry10 = remainder (y, 10)
sumc = add-single (rx10, ry10, c)
qsum10 = quotient (sumc, 10)
rsum10 = remainder (sumc, 10)

convert (x, y) = (x * 10) + y

{ Multiplication by 10 → left shift
Add a single digit number to a multiple of 10

3. Find the terminating clause

addc (0, 0, c) = c

```
(define (add x y) (addc x y 0))

(define (addc x y c)
  (cond [(and (= x 0) (= y 0)) c]
        [else (let* [(qx10 (quotient x 10))
                     (rx10 (remainder x 10))
                     (qy10 (quotient y 10))
                     (ry10 (remainder y 10))
                     (sumc (add-single-carry rx10 ry10 c))
                     (qsum10 (quotient sumc 10))
                     (rsum10 (remainder sumc 10))]
              (convert (addc qx10 qy10 qsum10) rsum10))]))

(define (add-single-carry x y c)
  (if (and (is-single x) (is-single y) (is-single c)) ;all operators have a
                                                      multi-argument version
      (+ x y c)
      (error "Operands should be single digit")))

(define (convert x y) (+ (* x 10) y))

(define (is-single x) (= (quotient x 10) 0))
```

This is function, If we use data before its definition, then error

Now define multiply:

```
  758
   42
 -----
 1516   →  758 × 2
3012    →  758 × 40
-----
31636
```

mult (758, 42) = 758 × 42

= 758 × 40 + 758 × 2

= 758 × 4 × 10 + 758 × 2

mult(758, 4) × 10 + 758 × 2 → multn1 (758, 2)

mult (x, y) = add (mult(x, qy10) * 10, multn1 (x, ry10))

where qy10 = quotient(y, 10)

ry10 = remainder(y, 10)

mult (x, 0) = 0

(multn1 758 2) = 758 x 2

= (750 + 8) x 2

= 75x2x10 + 8x2

= 75*2*.0 + 1*10 + 6

= (convert (multwc 75 2 1) 6)

Start with a carry of 0

multnic (758, 2, 0) = 758 * 2 + 0

= 750 x 2 + (8x2+0) → prodc

= 75x2x10 + 16

= 75x2x10 + 1x10 + 6 qp10, rp10

= (75 x 2 + 1) * 10 + 6

= convert(multnic (75, 2, 1), 6)

multnic (x, y, c) = convert(multnic(qx10, y, qp10), rp10)

where qx10 = ---

rx10 = ---

prodc = rx10 * y + c

qp10 = qutotient(prodc, 10)

rp10 = remainder (prodc, 10)

x y

mult (758, 42) = 758 × 42

= 758 × 40 + 758 × 2

= 758 × 4 × 10 + 758 × 2

= mult(758, 4) × 10 + 758 × 2

= mult (758, 4) * 10 + 758 * 2 + 0

x

qy10

multnlc (758, 2, 0)

ry10

multnlc (758, 2, 0) = 758 * 2 + 0

x y c = 750 * 2 + 8 * 2 + 0

= 75 * 2 * 10 + 16 prodc

= 75 * 2 * 10 + 1 * 10 + 6

= (75 * 2 + 1) * 10 + 6

= convert (multnlc (75, 2, 1), 6)

qx10 y pq10 pr10

```
(define (mult x y)
  (if (= y 0) 0
      (let* [(qy10 (quotient y 10))
             (ry10 (remainder y 10))]
        (add (* (mult x qy10) 10) (multn1c x ry10 0)))))

(define (multn1c x y c)   ;; y is single digit, c is carry
  (if (= x 0) c
      (let* [(qx10 (quotient x 10))
             (rx10 (remainder x 10))
             (prodc (add (mult-single rx10 y) c))
             (qp10 (quotient prodc 10))
             (rp10 (remainder prodc 10))]
        (convert (multn1c qx10 y qp10) rp10))))

(define (mult-single x y)
  (if (and (is-single x) (is-single y)) ;all operators have a multi-argument version
      (* x y)
      (error "Operands should be single digit")))
```

(gg bexp $bexp_1$ exp_2)

(cond [$bexp_1$ exp_1]
      [$bexp_2$, exp_2]
      ⋮
      [$bexp_n$, exp_n]

true → #t {Don't put parenthesis}
#f

(let* ([var_1 .exp_1] → scope of these variables is limited
       [var_2, exp_2] to exp only.
       ⋮
       [ ])
exp)

How much time is required for multiplication?

multiplc —

3 digit

458 × 5 + 6

45 × 5 × 10 + 8 × 5 + 6

= 45 × 5 × 10 + 4 × 10 + 6

= (45 × 5 + 4) * 10 + 6

(45 × 5 + 4) → 2 digit

T(3) = T(2) + c, or in general

T(n) = T(n-1) + c → calculations that do not depend on the length of the first number (quotient, remainder, convert)

T(0) = c'

T(n) = nc + c' - Within a constant factor of n

What about mult?

3 digits

458 * 256

= 458 * 25 *10 + 458 * 6

25: 2 digits; 458 * 6: multiplic

addition

$$\begin{cases} T(0) = c \\ T(n) = T(n-1) + c1 + c2*n + c3 + c4*n \end{cases}$$

T(n-1): mult; c1: quotient, remainder, *10; c2*n + c3: multiplic; c4*n: add

$$= T(n-1) + c_5 * n + c_6$$

$$T(n) = k_1 n^2 + k_2 n + k_3 \qquad O(n^2) \qquad \text{(Abstraction!)}$$

From now on we shall assume that our numbers are in binary form.

From now on assume that our numbers are represented in binary (though

Another method for multiplication (Al-Khwarizmi's method)

$$
\begin{aligned}
& 42 \times 84 \\
= \; & 21 \times 168 \\
= \; & 10 \times 336 + 168 \\
= \; & 5 \times 672 + 168 \\
= \; & 2 \times 1344 + 672 + 168 \\
= \; & 1 \times 2688 + 672 + 168 \\
= \; & 0 \times 5376 + 2688 + 672 + 168
\end{aligned}
$$

$$
\begin{array}{r}
2688 \\
\hline
2 \\
\hline
5376
\end{array}
$$

$27 = 16 + 8 + 2 + 1$

11011

$13 = 1101$

Division by 2 is right shift

Checking for oddness - rightmost bit is 1.

How to convert this to a program:

```
(define (ak-mult x y)
  (cond [(= 0 x) 0]
        [else (let [(ans (ak-mult (quotient x 2) (* y 2)))]
                (cond [(even? x) ans]
                      [#t (+ ans y)]))]))
```

variable

(else (+ ans y))]))

```
  42  x  84  ,  0
= 21  x  68  ,  0
= 10  x  336 ,  168
=  5  x  672 ,  168
=  2  x  1344,  840
=  1  x  2688,  840
=  0  x  5376,  3528
```

```
(define (ak-mult x y) (ak-helper x y 0))
```

multtr ⟶ tail recursion

```
(define (ak-helper x y c)
  (cond [(= 0 x) c]
        [(even? x) (ak-helper (quotient x 2) (* y 2) c)]
        [#t (ak-helper (quotient x 2) (* y 2) (add c y))]))
```

else

Division : Integer division

Given x and y, we want a pair of numbers q and r such that

$$x = q * y + r, \quad 0 \leq r \leq y$$

Example division (25, 7)

q, r : $25 = q * 7 + r \quad 0 \leq r < y.$

consider division (12, 7), answer is (1, 5)

$$12 = 1 * 7 + 5$$
$$2 * 12 + 1 = 2 * 1 * 7 + 2 * 5 + 1$$
$$25 = 2 * 7 + 11$$
$$= (2 + 1) * 7 + 4$$

This suggest a function for division.

division (x, y) = if $r'' \geq y$ Then $(2q' + 1, r'' - y)$ else $(2q', r'')$

where (q', r') = division (quotient $(x, 2), y)$

r'' = if (odd x) Then $2 * r' + 1$ else $2 * r'$

Modular Arithmetic. RSA - scheme for encoding information

— Operate within numbers of fixed length.

— $x \bmod n$ The remainder r when x is divided by n

i.e. $x = qn + r \quad , \quad 0 \leq r < n$

Make a distinction between:

(i) The mod operator as defined above

(ii) A congruence relation induced by the operator

Congruent ↓ equivalent, for our purpose.

$1 \bmod 3$, $4 \bmod 3$, $7 \bmod 3$ | $-2 \bmod 3$, $-5 \bmod 3$, $-8 \bmod 3$

$x \equiv_n y \iff n \mid x-y$. ($n$ divides $x-y$)

$x \equiv n\ y$

Modular arithmetic

$(x +_n y) = (x+y) \bmod n$

$(x *_n y) = (x * y) \bmod n$

- Let → value of variables from outside
  let* → value within it or else from outside