Chapter 0_02

Introducing Foundation

Body of Knowledge Coverage:
Basis Analysis (AL)

- Basis Analysis (AL)
  - Asymptotic Analysis, empirical measurement.
  - Differences among best, average, and worst case behaviors of an algorithm.
  - Complexity classes, such as constant, logarithmic linear, quadratic, and exponential.
  - Recurrence Relations and their solutions.
  - Time and space trade-offs in algorithms.

Outlines:

- Sum of any three single-digit numbers is at most two digits long.
- Number of digits needed to represent the number $N \geq 0$ in base b.
- Algorithm for determining *number of binary digits needed* in the binary representation of a positive decimal integer n.
- Algorithm analysis framework
- How much does the size of a number change, when change base?

Number Theory Review

A basic property of numbers in any base $b \geq 2$:

The sum of any three single-digit numbers is at most two digits long.

Example 0.1:

For decimal numbers in base 10: $9 + 9 + 9 = 27_{10}$

For binary numbers in base 2: $1 + 1 + 1 = 11_2$

For hexadecimal numbers in base 16: $F + F + F = 1111 + 1111 + 1111$

$= 2D_{16}$

$$
\begin{array}{r}
1111 \\
\underline{+1111} \\
0001\ 1110 \\
\underline{+1111} \\
0010\ 1101 \\
0010\ 1101 = 2D_{16}
\end{array}
$$

For Octal numbers in base 8: $7 + 7 + 7 = 111 + 111 + 111$

$= 25_8$

$010\ 101 = 25_8$

Note that $D_{16} = 1101_2 = 13_{10}$

Number Theory Review

How many digits (k) are needed to represent the number $N \geq 0$ in base b?

With k digits in base b, there are numbers $\{N \mid b^{k-1} \leq N < b^k$ and N has k number of digits$\}$.

Example 0.2:

- For b = 10 (decimal) and if k = 3, then

  $100 = 10^2 \leq N \leq 10^3 - 1 = 1000 - 1 = 999_{10}$

- For b = 2 (binary) and if k = 8, then

  $2^7 \leq N \leq 2^8 - 1$, where $2^7 = 128_{10} = 1000\ 0000_2 = 80_{16}$ and

  $2^8 - 1 = 256 - 1 = 255_{10}$ (i.e., $1111\ 1111_2 = FF_{16}$).

- For b = 16 (hexadecimal) and if k = 4, then

  $F000 = 4096 = 16^3 \leq N \leq 16^4 - 1 = 65536 - 1 = 65535 = FFFF_{16}$

Example of Binary(n)

- Design an algorithm:
  - finding *the number k of binary digits* in the binary representation of a positive decimal integer n.
  - Analysis: Since $2^{k-1} \le n < 2^k$, then $\log_2 2^{k-1} \le \log_2 n < \log_2 2^k$.
    $k-1 \le \log_2 n < k$. $k-1 \le \lfloor \log_2 n \rfloor < k$. $k \le \lfloor \log_2 n \rfloor + 1 < k + 1$.
- For example: it needs
  - one binary digit to represent 0 or 1 (<u>0</u> or <u>1</u>);
  - two binary digits to represent 2 (<u>10</u>) through 3 (<u>11</u>),
  - three binary digits to represent 4 (<u>100</u>) through 7 (<u>111</u>), and
  - four binary digits to represent 8 (<u>1000</u>) through 15 (<u>1111</u>),
  - etc.

Example of Binary(n)

Design an algorithm for finding *the number k of binary digits* in the binary representation of a positive decimal integer n.

Let k be count.

```
Algorithm Binary(n)
    Input:   A positive decimal integer n
    Output: The number of binary digits in n’s binary representation
    count ← 1; //let k be count.
    while (n > 1) do {
          count ← count + 1;
          n ← ⌊n/2⌋; }
    return count;
```

Number of executions of > is 4 times.
Number of executions of + is 3 times.

(n > 1)	15 > 1	7 > 1	3 > 1	1 > 1 (false)
k = 1	k = 2	k = 3	k = 4	exitWhile at n=1
n =15	$\lfloor 15/2 \rfloor$	$\lfloor 7/2 \rfloor$	$\lfloor 3/2 \rfloor$	return count = 4

Requires four bits to encode 15 as 1111.

Time efficiency is count = $\lfloor \log_2 n \rfloor + 1$, for a large k, $2^k \leq n$.
It can compute via $T(n) = T(\lfloor n/2 \rfloor) + 1$, where $n = 2^k$.

Analysis Framework

1. Measuring an input's size:

- The input for this algorithm is an integer n.
- The input size is $\lfloor \log_2 n \rfloor + 1$.
    - Define the *input size* as the number of symbols (i.e., binary digits (bits)) used for the encoding a positive integer n.
- Example:
    - If n = 15, the input size is $\lfloor \log_2 15 \rfloor + 1 = 3 + 1$ bits. i.e., 15 in binary representation is 1111_2
    - If n = 16, the input size is $\lfloor \log_2 16 \rfloor + 1 = 4 + 1$ bits. i.e., 16 in binary representation is 10000_2

Analysis Framework

```
Algorithm Binary(n)
    …
    while (n > 1) do {
        count ← count + 1;
        n ← ⌊n/2⌋; }
    …
```

2. Units for measuring running time
   - The most frequently executed operation:
     - The comparison $n > 1$
       - determines whether the loop's body is to be executed.
     - The number of times the comparison is to be executed = the number of repetitions of the loop's body + 1.

   - For the loop's variable (i.e., n):
     - The value of n is about halved on each repetition of the loop (i.e., $n \leftarrow \lfloor n/2 \rfloor$), where $2^{k-1} \le n < 2^k$ and k is the number of times dividing by 2.
     - The number of times the loop to be executed would be about $\lfloor \log_2 n \rfloor + 1$.
       - [i.e., $2^{k-1} \le n < 2^k$, then $(k-1) \log_2 2 \le \log_2 n < k \log_2 2$,
         $k-1 \le \log_2 n < k$]

$2^{k-1} \le n < 2^k$
If $k = 4$ then
1000 to 1111

Example 0.6:

- The exact formula for the number of times, $C(n)$, the comparison $n > 1$ to be executed is:

$$C(n) = k + 1, \text{ if } n = 2^{k-1}, \text{ then } \log_2 n = \log_2 2^{k-1}$$

$$= (k-1)\log_2 2 = k + 1$$

$$= \lfloor \log_2 n \rfloor + 1, \text{ where } 2^{k=1} \leq n < 2^k$$

$$= \Theta(\log_2 n)?$$

- The number of bits in the binary representation of n is

$$k + 1 = \lfloor \log_2 n \rfloor + 1 \text{ bits}$$

```
Algorithm Binary(n)
    …
    while (n > 1) do {
        count ← count + 1;
        n ← ⌊n/2⌋; }
    …
```

Note: For $2^{k-1} \le n < 2^k$, the number of bits k for representing n is

$$k = \lfloor \log_2 n \rfloor + 1$$

The number of bits for representing $8 \le n < 16$ are:

$\lfloor \log_2 8 \rfloor + 1 = 3 + 1$, where $8 = 2^3$	1 0 0 0
$\lfloor \log_2 9 \rfloor + 1 = 3 + 1$	1 0 0 1
$\lfloor \log_2 10 \rfloor + 1 = 3 + 1$	
…	
$\lfloor \log_2 15 \rfloor + 1 = 3 + 1$	1 1 1 1
$\lfloor \log_2 16 \rfloor + 1 = 4 + 1$ where $16 = 2^4$	1 0 0 0 0

We can show $\lfloor \log_2 n \rfloor + 1 = \lceil \log (n + 1) \rceil$.

How much does the size of a number change when we change base?

How much does the size of a number change, when we change base?

- The rule of converting logarithms from base a to base b:

$$log_b N = \frac{log_a N}{log_a b}.$$ That is, $log_a N = log_a b\,(log_b N)$.

- The size of integer N in base a is the same as a constant factor $log_a b$ of its size in base b.

- Example: Consider $256_{10} = 100_{16} = 1\ 0000\ 0000_2$.

$$\log_{16} 256 = \frac{\log_2 256}{\log_2 16}.$$ i.e., $\log_2 256 = \log_2 16\,(\log_{16} 256)$.

$$\log_{16} 16^2 = \frac{\log_2 2^8}{\log_2 2^4}$$

$$2 = 2$$

- This shows that it requires 3 characters to represent 256 in base 16, which is 100_{16} and 9 bits binary representation $1\ 0000\ 0000_2$.

- How long does the addition algorithm take to add two given numbers.

- Multiplication: Left-shifting is a quick way to multiply by the base 2.
- How long does the multiplication algorithm takes?

For any problem-solving, we always consider the following questions:

Construct an addition algorithm of two numbers in any base.

Analysis

- By the basic property of numbers in any base,
  - each sum of two single-digit numbers is a two-digit number;
  - the carry is always a single digit; and
  - at any given step, three single-digit numbers are added.
    - sum of three single-digit numbers is a two-digit number.

Algorithm

- Align their right-hand ends, and then
- ***perform a single right-to-left pass*** in which
  - the sum is computed digit by digit,
  - maintaining the overflow as a carry.

Q: Given two binary numbers x and y, how long does our algorithm take to add them?

How long does our algorithm take to add two given binary numbers x = 53_{10} and y = 35_{10}?

Carry	1			1	1	1		
		1	1	0	1	0	1	(53)
		1	0	0	0	1	1	(35)
	1	0	1	1	0	0	0	(88)

Total running time as a function of the size of the input: the number of bits of x or y.

Analysis: Total running time as a function of the size of the input:

the number of bits for representing two integers x and y.

Assume that each of x and y are of n bits long.

- Adding two n-bit numbers requires n operations, disregarding at least read them and write down the answer.
- The sum of x and y is n+1 bits at most.
- Adding three binary digits requires a fixed amount of time.
- The total running time for the addition algorithm is of the form $c_0 + c_1 n$, where c_0 and c_1 are some constants.
- It is linear. The running time is $O(n)$.

Is there a faster algorithm?

The addition algorithm is optimal, up to multiplicative constants.

- The total running time for the addition algorithm is of the form $c_0 + c_1 n$, where c_0 and c_1 are some constants.

Multiplication and Division

The *grade-school algorithm* for multiplying two numbers x and y is:

- create an array of intermediate sums,
  - each representing the product of x by a single digit of y.
- These values are appropriately left-shifted and then added up.
- For example: multiply 13 by 11.

	1	3	
*	1	1	
	1	3	(multiply 13 by 1)
1	3		(multiply 13 by 1, shifted once)
1	4	3	

An array of intermediate sums

Left-shifting (for multiplication) is a quick way to multiply by the base-Reason:

- Given two integers 13 and 2, 13 x 2 can be written in binary representation as 1101 x 10. The result 26 (11010 in bit representation) can be obtained by left-shifting one-bit position 1101 and packing a 0 on the rightmost bit to form 11010.

	1	1	0	1	(multiplicand)
x			1	0	(multiplier)
	0	0	0	0	(multiply 1101 by 0)
1	1	0	1	0	(multiply 1101 by 1, left-shift once)
1	1	0	1	0	

- In binary multiplication, each intermediate row is
  - *either filling with 0's* if the multiplier's bit is 0, *or*
  - *copying* the multiplicand if *the multiplier's bit is 1.*
  - right-align the multiplicand with the multiplier's bit,
    - using left-shifted an appropriate amount of times with packing 0 on the rightmost bit(s) .

- Likewise, the effect of a right shift (for division by 2) is to divide by the base, rounding down if needed. – Integer Division
  - An example: 13/2 is 1101 ÷ 10.
    - The result is $\lfloor 13/2 \rfloor = 6$ (0110 in bit representation) can be obtained by right-shift one-bit position 1101 and pack a 0 on the leftmost bit to form 0110, which is 6. (integer division).
  - Example: 13/4, which is (13/2)/2.
    - Allows to shift-right 1101 twice and packs 00 on the leftmost (i.e., significant) bits to obtain 0011, which is equal to 3. That is 13/2 = 6, and then 6/2 = 3.
  - Example: 13/8 = ((13/2)/2)/2.
    - Allows to shift-right thrice 1101 and packs 000 on the significant bits to obtain 0001, which is equal to 1. Since $\lfloor (13/2)/2 \rfloor = 3$, then $\lfloor 3/2 \rfloor = 1$.
  - Example: 13/16 = (((13/2)/2)/2)/2.
    - Allows us to shift right four times and pack 0000 on the significant bits to obtain 0000, which is equal to 0. Continue from above, $\lfloor 1/2 \rfloor = 0$.

How long the grade-school algorithm takes:

	1	3
x	1	1
	1	3
1	3	
1	4	3

Consider 13 x 11, which is (1101 x 1011)

The multiplication would proceed as follows.

				1	1	0	1	
			x	1	0	1	1	
				1	1	0	1	(1101 times 1)
			1	1	0	1	0	(1101 times 1, shift once)
		0	0	0	0	0	0	(1101 times 0, shift twice)
+	1	1	0	1	0	0	0	(1101 times 1, shift thrice)
1	0	0	0	1	1	1	1	(binary for 143)

n-1 times row additions

- Let x and y are both n bits.
- There are n intermediate rows,
  - with lengths of up to 2n bits (take the shifting into account).
- The total time is taken to add up these rows,
  - *doing two numbers* (per two rows) at a time is:

$$\underbrace{O(n) + O(n) + \ldots + O(n)}_{n - 1 \text{ times}},$$

[Sum of each two intermediate rows requires O(n).]

[Requires n-1 times of 2 number addition for n rows]

[i.e., $(n-1) * O(n) = O(n^2)$]

which is $O(n^2)$, quadratic in the size of the inputs:
still polynomial but much slower than addition.

- Is there a faster algorithm for multiplication?
- Al Khwarizmi's Algorithm: (Persian mathematician)
- Multiplication à la Français - function multiply(x, y)
  - A recursive algorithm
  - Algorithm analysis
- Multiplication à la Russe - a nonorthodox algorithm for multiplying two positive integers.

Is there a faster algorithm for multiplication?

`Al Khwarizmi’s Algorithm:`

Multiply two decimal numbers x and y:

- Repeat the following until the first number y gets down to 1:
  1. integer-divide the first number y (multiplier) by 2, and
  2. double the second number x (multiplicand).
- Then strike out all the row in which the first number y is even (why?),

  It is because the 0 digit in y.
- and add up what remains in the second column.

Example 0.7: Let y = 11 (1011) and x = 13 (1101).

x * y = 13 * 11 = 1101 * 1011.

The bit 0 in y, it yield 0000 for an intermediate row.

The algorithm is:

if y is odd, then x + 2z

else 2z, where z = x * (y/2) and (y/2) is an integer division.

$$y * x = \begin{cases} x + 2\,(x * y/2), & \text{if } y \text{ is odd} \\ 2\,(x * y/2), & \text{if } y \text{ is even.} \end{cases}$$

(why y/2?)

y	x									x * y = 1101 * 1011
11	13						1	1	0	1
5	26	(why x*2?)				1	1	0	1	0
2	52	(strike out)			0	0	0	0	0	0
1	104	(why x*2?)		1	1	0	1	0	0	0
	143	(answer)	1	0	0	0	1	1	1	1

$$x * y = \begin{cases} x + 2\,(x * y/2), & \text{if } y \text{ is odd} \\ 2\,(x * y/2), & \text{if } y \text{ is even.} \end{cases}$$

5 * 26 = 26 + 2(26 * 5/2)
= 26 + 26 * 4

2 * 52 = 2(52 * 2/2)

4 * 52 = 2(52 * 4/2) Facts!

13 * 11 = 13 + 2(13 * 11/2) = 13 + (2 * 13 *5) since y is 11, an odd number.

= 13 + (**26 * 5**) = 13 + (26 + 2(26 * 5/2)), since y = 5, an odd number.

= 13 + (26 + (52 * 5/2))

= 13 + (26 + **(52 * 2)**)

= 13 + (26 + (2 (52 * 2/2))) since y = 2, an even number.

= 13 + (26 **+ (104 *1)**)

= 13 + (26 + 104)

= 13 + 130

= 143.

y	x
11	13
5	26
2	52
1	104
	143

Reason:

$$y * x = \begin{cases} x + 2\,(x * y/2), & \text{if } y \text{ is odd} \\ 2\,(x * y/2), & \text{if } y \text{ is even.} \end{cases}$$

$$\begin{aligned} X * Y &= (X * Y) * \frac{2}{2} \\ &= 2 * X * \frac{Y}{2} \\ &= 2 * X * \frac{Y-1+1}{2}, \text{ if Y is odd} \\ &= 2 * X * \left(\frac{Y-1}{2} + \frac{1}{2}\right) \\ &= 2 * X * \frac{Y-1}{2} + 2 * X * \frac{1}{2} \\ &= 2 * X * \frac{Y-1}{2} + X \\ &= X + 2 * X * \lfloor y/2 \rfloor \\ &= X + 2(X * \lfloor y/2 \rfloor) \end{aligned}$$

$$\begin{aligned} X * Y &= (X * Y) * \frac{2}{2} \\ &= 2 * \left(X * \frac{Y}{2}\right), \text{ if Y is even.} \end{aligned}$$

$$y * x = \begin{cases} x + 2\,(x * y/2), & \text{if } y \text{ is odd} \\ 2\,(x * y/2), & \text{if } y \text{ is even.} \end{cases}$$

Why this system does in this way?

$11 * 13 = (1+ 10) * 13 = 13 + (10 * 13) = 13 + (10/2 * 13*2)$

$= 13 + (5 * 26)$

$= 13 + ((1 + 4) * 26)$

$= 13 + (26 + (4 * 26)) = 13 + (26 + (4/2 * 26*2))$

$= 13 + (26 + (2 * 52))$

$= 13 + (26 + (2/2 * 52*2))$

$= 13 + (26 + (1 * 104))$

$= 13 + (26 + ((1 + 0) * 104)) = 13 + (26 + (104 + (0* 104)))$

$= 13 + (26 + (104)) = 143$

y	x
11	13
5	26
2	52
1	104
	143

Al Khwarizmi's Algorithm resolves the product of two integers as a process of halving one integer by 2, and doubling the other integer by 2 and adding this integer if needed. Why multiply by 2? $O(n)$? $O(n^2)$? *It is* $O(n^2)$.

Al khwarizmi's algorithm is a fascinating mixture of decimal and binary.

Integer division

$$y * x = \begin{cases} x + 2\,(x * y/2), & \text{if } y \text{ is odd} \\ 2\,(x * y/2), & \text{if } y \text{ is even.} \end{cases}$$

Al khwarizmi's algorithm

The same algorithm can be rewritten in different way, based on the following rule:

$$x * y = \begin{cases} 2(x * \lfloor y/2 \rfloor) & \text{if } y \text{ is even} \\ x + 2(x * \lfloor y/2 \rfloor) & \text{if } y \text{ is odd} \end{cases}$$

- Shift X to left by |y| bit packing with 0’s. This need O(|y|).
- Add the intermediate results obtained from left shift by x at most |y|-1 times.
- That is O(|y| * (|x| + |y|)). For both X and Y has n bits, then O(n^2).

y	**x**		y	x	x * y = 1101 * 1011							
1011	**1101**		11	13					1	1	0	1
101	**11010**		5	26				1	1	0	1	0
10	~~110100~~	(s	2	52			0	0	0	0	0	0
1	**1101000**		1	104		1	1	0	1	0	0	0
	10001111	(a		143	1	0	0	0	1	1	1	1

Example 0.8: Let $x = 13$ and $y = 16$. Find $x * y$. ($13 = 1101$, $16 = 10000$)

($13 = 1101$, $16 = 10000$)

Computed by hands

y	x		1101*10000
16	13	(strike out)	0000
8	26	(strike out)	00000
4	52	(strike out)	000000
2	104	(strike out)	0000000
1	208		11010000
	208	(answer)	11010000

The use of the algorithm, we have

$13 * 16 = 13 * 2 * 16/2 = 2(13 * 8)$

$13 * 8 = 13*2 * 8/2 = 2(13 * 4)$

$13 * 4 = 13*2 * 4/2 = 2(13 * 2)$

$13 * 2 = 13 *2 * 2/2 = 2(13 * 1)$

$13 * 1 = 13 * (1 + 0) = 13 + 2(13*0/2)$.

$$
\begin{aligned}
13 * 16 &= 2(13 *8) = 2(2(13 * 4)) \\
&= 2(2(2(13*2))) \\
&= 2(2(2(2(13*1)))) \\
&= 2(2(2(2(13*(1+0))))) \\
&= 2(2(2(2(13+ 2(13*0/2))))) \\
&= 2(2(2(2(13)))) \\
&= 2(2(2(26))) \\
&= 2(2(52)) \\
&= 2(104) = 208
\end{aligned}
$$

Example 0.8: Let x = 13 and y = 16. Find x * y. (13 = 1101, 16 = 10000)
(13 = 1101, 16 = 10000)

Computed by bit-representations

y	x		1101*10000
10000	~~1101~~	(strike out)	0000
1000	~~11010~~	(strike out)	00000
100	~~110100~~	(strike out)	000000
10	~~1101000~~	(strike out)	0000000
1	11010000		11010000
	11010000	(answer 208)	11010000

The use of the algorithm, we have

13 * 16 = 13 *2 * 16/2 = 2(13 * 8)

13 * 8 = 13*2 * 8/2 = 2(13 * 4)

13 * 4 = 13*2 * 4/2 = 2(13 * 2)

13 * 2 = 13 *2 * 2/2 = 2(13 * 1)

13 * 1 = 13 * (1 + 0) = 13 + 2(13*0/2).

13 * 16 = 2(13 *8) = 2(2(13 * 4))

= 2(2(2(13*2)))

= 2(2(2(2(13*1))))

= 2(2(2(2(13*(1+0)))))

= 2(2(2(2(13+ 2(13*0/2)))))

= 2(2(2(2(13))))

= 2(2(2(26)))

= 2(2(52))

= 2(104) = 208

Example 0.9: Let x = 13 and y = 38. Find x * y. (13 = 1101, 38 = 100110)

Computed by bit-representations

y	x		1101*100110
100110	~~1101~~	(strike out)	0000
10011	11010		11010
1001	110100		110100
100	~~0000000~~	(strike out)	0000000
10	~~00000000~~	(strike out)	00000000
1	110100000		110100000
	111101110	(answer 494)	111101110

The use of the algorithm, we have

$13 * 38 = 2(13 * 38/2) = 2(13 * 19)$

$13 * 19 = 13 *(1 + 18) = 13 + (13 * 18)$

$= 13 + 2(13 * 18/2) = 13 + 2(13 * 9)$

$13 * 9 = 13 *(1 + 8) = 13 + 2(13 * 8/2)$

$= 13 + 2(13 * 4)$

$13 * 4 = 2(13 * 4/2) = 2(13 * 2)$

$13 * 2 = 2(13 * 2/2) = 2 (13*1)$

$13 * 1 = 13 *(1+0) =13 + 2 (13*0/2) = 13$

$13 * 38 = 2(13 * 19) = 2(13 + 2(13 * 9))$

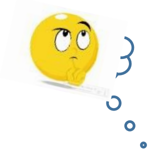
$= 2(13 + 2(13 + 2(13 * 4)))$

$= 2(13 + 2(13 + 2(2(13 * 2))))$

$= 2(13 + 2(13 + 2(2(2(13 * 1)))))$

Figure 1.1: Multiplication à la Français -
A recursive algorithm which directly implement this rule :

$$x * y = \begin{cases} 2(x * \lfloor y/2 \rfloor), & \text{if } y \text{ is even} \\ x + 2(x * \lfloor y/2 \rfloor), & \text{if } y \text{ is odd} \end{cases}$$



function multiply(x, y)

Input: Two n-bit integers x and y, where $y \geq 0$

Output: Their product

if (y = 0) then return 0;

z := multiply (x, $\lfloor y/2 \rfloor$);

if (y is even) then return 2z

else return x + 2z;

Figure 1.1 Multiplication à la Français

Using this algorithm, we have (example 0.9)

13 * 38 = 2(13 * 19)

13 * 19 = 13 + 2(13 * 9)

13 * 9 = 13 + 2(13 * 4)

13 * 4 = 2(13 * 2)

13 * 2 = 2(13 * 1) = 2 * 13 = 26,

13 * 1 = 13 + 2 (13* $\lfloor 1/2 \rfloor$)

= 13 + 2 * 0 = 13

13 * 38 = 2(13 * 19)

= 2(13 + 2(13 * 9))

= 2(13 + 2(13 + 2(13 * 4)))

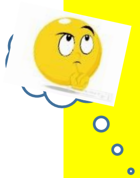
= 2(13 + 2(13 + 2(2(13 * 2))))

= 2(13 + 2(13 + 2(2(2(13 * 1)))))

Analysis of an algorithm:

- *Is this algorithm correct?*
- *How long does the algorithm take?*
- *Can we do better?*

```
function multiply(x, y)
    Input: Two n-bit integers x
           and y, where y ≥ 0
    Output: Their product

    if  (y = 0)  then return 0;
    z :=  multiply (x, ⌊ y/2 ⌋);
    if (y is even) then return 2z
              else return x + 2z;
```



Is this algorithm correct?

- *Does algorithm behave what it intends to do?*
  - *Given input and output specifications, will algorithm produce ouput_data that satisfies the output specification for all the input_data satisfies the input specification?*
- *Will algorithm halt?*

It is transparently correct; It also handles the base case (y = 0).

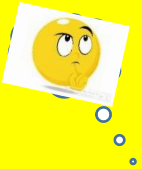
```
function multiply(x, y)
    Input: Two n-bit integers x
           and y, where y ≥ 0
    Output: Their product

    if  y = 0  then return 0;
    z :=  multiply (x, ⌊ y/2 ⌋);
    if y is even then return 2z
              else return x + 2z;
```

$$x * y = \begin{cases} 2(x * \lfloor y/2 \rfloor), & \text{if } y \text{ is even} \\ x + 2(x * \lfloor y/2 \rfloor), & \text{if } y \text{ is odd} \end{cases}$$

How long does the algorithm take?

- The function for multiplying two n-bit integers, terminate after n recursive calls, because y is halved ($\frac{y}{2}$) at each call.
  - i.e., the number of bits of y is decreased by one (i.e., right-shift once).
- Upon return from each recursive call, requires a total of O(n) bit operations, which are as follows.
  - A division by 2 (using right-shift) for $\lfloor y/2 \rfloor$;
  - a test for even/odd (looking up the rightmost bit either 0 or 1);
  - a multiplication by 2 (using left-shift); and
  - one addition if y or $\lfloor y/2 \rfloor$ is odd.
- $T(n) = n*O(n) = n*(c_1n+c_2) = c_1n^2+c_2n = O(n^2)$.

Therefore, the total time taken is thus $O(n^2)$.



```
function multiply(x, y)
    Input: Two n-bit integers x
           and y,  where y ≥ 0
    Output: Their product

    if  y = 0  then return 0;
    z :=  multiply (x, ⌊y/2⌋);
    if y is even then return 2z
         else return x + 2z;
```

Shift right n times for n-bit y.
Therefore, n recursive calls.

if y is even then return 2z
else return x + 2z;
This takes linear time to do it for each round.

Prove $T(n) = O(n^2)$

$T(n) = T(\lfloor n/2 \rfloor) + c(n)$

$T(1) = c_0$ (assume $c_0 = 1$);

Solution: (need to check the correctness of the following)

Let $n = 2^k$.

$T(n) = T(2^k) = T(2^{k-1}) \ldots {}^{k}$

$= T(2^{k--2}) + 2^{k} \ldots 2^{k}$

$=$

(2^k)

$k = i$

$=$

The algorit…

function multiply(x, y)
Input: Two n-bit integers x and y, where $y \geq 0$
Output: Their product

if $y = 0$ then return 0;
$z := \text{multiply}(x, \lfloor y/2 \rfloor)$;
if y is even then return 2z
else return x + 2z;

Let n = 1.

$$\text{multiply}(x, y) = \begin{cases} \text{return } 0, \text{ if } y = 0; \\ \text{return } x, \text{ if } y = 1; \\ \text{since } y = 1, \\ z := \text{multiply}(x, \lfloor y/2 \rfloor) \\ \quad = \text{multiply}(x, 0) = \text{returns } 0, \text{ since } y = 0. \\ \text{return } x + 2*0 = x, \text{ since } y = 0. \end{cases}$$

Conclusion: $T(n = 1\text{bit}) = c_0$

Prove $T(n) = O(n^2)$

$T(n) = T(\lfloor n/2 \rfloor) + c(n)$

$T(1) = c_0$ (assume $c_0 = 1$);

Solution: (check the correctness of the following)

Let $n = 2^k$.

$$\boxed{T(n) = T(\tfrac{n}{2}) + c(n)}$$

$$\boxed{T(2^k) = T(2^{k-1}) + (2^k)}$$

$$
\begin{aligned}
T(n) = T(2^k) &= T(2^{k-1}) + 2^k \\
&= T(2^{k-2}) + 2^{k-1} + 2^k \\
&= \ldots \\
&= T(2^{k-i}) + (2^{k-i+1}) + (2^{k-i+2}) + \ldots + (2^{k-3}) + (2^{k-2}) + (2^{k-1}) + (2^k) \\
&= T(2^{k-k}) + (2^{k-k+1}) + (2^{k-k+2}) + \ldots + (2^{k-3}) + (2^{k-2}) + (2^{k-1}) + (2^k),\ k = i \\
&= T(2^{k-k}) + (2^{k-k+1}) + (2^{k-k+2}) + \ldots + (2^{k-3}) + (2^{k-2}) + (2^{k-1}) + (2^k), \\
&= 1 + (2^1) + (2^2) + \ldots + (2^{k-3}) + (2^{k-2}) + (2^{k-1}) + (2^k), \\
&= (2^{k+1} - 1) \\
&= 2n - 1 = O(n) \text{ for each recursive call}
\end{aligned}
$$

The algorithm will take n calls. For each call's return, it requires $O(n)$ for addition. Therefore $O(n^2)$.

```
function multiply(x, y)
      Input: Two n-bit integers x and y,
                  where y ≥ 0
      Output: Their product

      if  y = 0  then return 0;
      z :=  multiply (x, ⌊ y/2 ⌋);
      if y is even then return 2z
            else return x + 2z;
```

n bits integer requires n times of right shifts, that is $\frac{n}{2}$. Therefore it takes n calls.

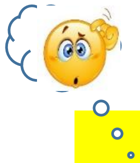
Can we do better?

- We can do significantly better. (See Chapter 02)

```
function multiply(x, y)
    Input: Two n-bit integers x and y,
                    where y ≥ 0
    Output: Their product

    if  y = 0  then return 0;
    z :=  multiply (x, ⌊y/2⌋);
    if y is even then return 2z
              else return x + 2z;
```

Multiplication ả la Russe

- Consider the multiplication ả la Russe,
  - a nonorthodox algorithm for multiplying two positive integers, x and y.



  - also called the Russian Peasant Method.
- Let x and y be positive integers.
- Compute the product of x and y using:

$$x * y = \begin{cases} \frac{y}{2} * 2x & \text{if y is even} \\ \frac{y-1}{2} * 2x + x & \text{if n is odd} \end{cases}$$

Al khwarizmi’s method:

$$x*y = \begin{cases} 2(x * \lfloor y/2 \rfloor) & \text{if y is even} \\ x + 2(x * \lfloor y/2 \rfloor) & \text{if y is odd} \end{cases}$$

Compute the product x * y, where y and x are positive integers.

- Compute product **x * y** either recursively or iteratively**.**
- The difference between the Russian Peasant Method and *Al Khwarizmi's algorithm (coded as* Multiplication à la Français) is that
  - integer (even) division, and
  - reduce the value of y by 1 if y is an odd number.

Let's measure the instance size by the value of y.

If y is even, an instance of half the size has to deal with y/2 or $\lfloor y/2 \rfloor$:

$y * x = (y/2) * 2x.$ (*Multiplication ả la Russe*)

$= (2x * \lfloor y/2 \rfloor)$ if y is even (Al khwarizmi's)

If y is odd, we have

$y * x = ((y - 1)/2) * 2x + x.$ (*Multiplication ả la Russe*)

$= x + 2(x * \lfloor y/2 \rfloor)$ if y is odd (Al khwarizmi's)

Using these formula, and the trivial case of $1 * x = x$ to stop.

Example 0.10: Compute 38 * 13

$$38 * 13 = \frac{38}{2} * 2 * 13 = 19 * 26$$

$$= \frac{19-1}{2} * 2 * 26 + 26 = 9 * 52 + 26$$

$$= (\frac{9-1}{2} * 2 * 52 + 52) + 26 = 4 * 104 + 52 + 26$$

$$= \left(\frac{4}{2} * 2 * 104\right) + 52 + 26 = 2 * 208 + 52 + 26$$

$$= \left(\frac{2}{2} * 2 * 208\right) + 52 + 26 = 1 * 416 + 52 + 26$$

$$= 494$$

y	**x**		
38	13		
19	26		38 is even
9	52	(+26)	19 is odd
4	104	(+52)	9 is odd
2	208		4 is even
1	416		2 is even
	494	+26+52+416	

What is the time efficiency class of Russian peasant multiplication?

The total time taken is thus $O(n^2)$. [Prove it.]

Example 0.11

An example of computing 50 * 65 with this algorithm is given in Figure:
All the extra addends shown in the parentheses in the Figure are in the rows with odd values in the first column. Apply the Russian Peasant Method and Al Khwarizmi's algorithm yielding the following results:

y	**x**	
50	65	
25	130	since 50 is even
12	260	(+130), since 25 is odd
6	520	since 12 is even
3	1040	since 6 is even
1	2080	(+1040), since one is odd
	3250	+(130 + 1040)=3250

y	**x**	
50	65	strike-out
25	130	
12	260	strike-out
6	520	strike out
3	1040	
1	2080	
	3205	

the Russian Peasant Method

Al Khwarizmi's algorithm

There we can find the product by simply adding all the elements in the **x** column that have an odd number in the **y** column. (See figure in below)

y	**x**	
50	65	
25	130	130
12	260	
6	520	
3	1040	1040
1	2080	2080
		3250

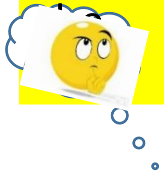
The algorithm involves only the simple operations of halving, doubling, and adding.

It also leads to very fast hardware implementation since doubling and halving of binary numbers can be performed using the left and right shifts, respectively, which are among the most basic operations at the machine level.

Division

Proof of Program Correction.

The iteration version of division is as follows: Based on $x = q * y + r$.



```
function divide(x, y)                    //what is input size?
```

Input: Two n-bit integers x and y, where $x \geq 0$, $y \geq 1$.

Output: The quotient and remainder of x divided by y.

//if $x = 0$, then return $(q, r) := (0, 0)$; $(r \geq y)$ takes care it

`q := 0;  r := x;` //assuming $x \leq 2^n - 1$, which is n bits long.

`while (r ≥ y) do` // takes 2^n iterations for the worse case, says x/1, where

`{` // x is n bits long with a max value $2^n - 1$.

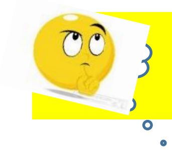
`q := q + 1;`

`r := r - y};` // O(n) for each $r - y$, where $r + (-y)$ are n bits addition, where x, y are n

// bits long, .

`return (q, r);`

This takes linear time O(n) for each iteration. Time efficiency for the worse case is exponential, $O(n\ 2^n)$. That is, $2^n\ O(n) = 2^n\ (c_0 + c_1\ n) = O(n\ 2^n)$. Not $O(n^2)$



The iteration version of division is as follows:

```
function divide(x, y)
```

Input: Two n-bit nonnegative integers x and y, where $x \geq 0$, $y \geq 1$.

Output: The quotient q and remainder r of x divided by y.

```
q := 0;  r := x;
```

[Repeatly substract y from r (i.e., x) until x number less than y is obtained. Add 1 to q each time y is subtracted. $0 \leq x - y - y - \ldots - y = x - yq < y$.]

```
while (r ≥ y) do
```

// takes 2^n *iterations* for the worse case. Let use $n = 2^n$, for
// the sake of simplicity in the correctness proof.

```
    { q := q + 1;
      r := r - y};
```

// O(n) for *each* r – y, where y is n bits long.

[After execution of the while loop, x = y q + r.]

```
return (q, r);
```

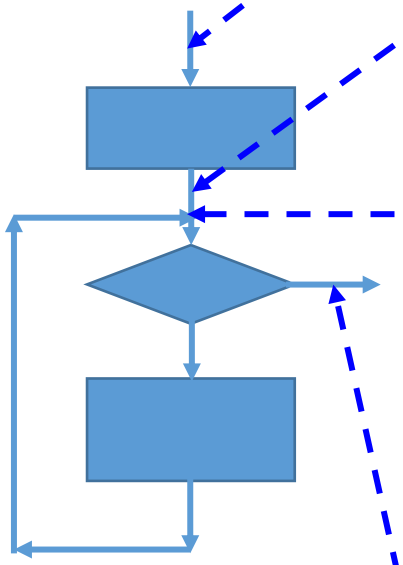
This takes linear time O(n) for each iteration. Time efficiency is $O(n2^n)$.

Correctness of the Division Algorithm

```
function divide(x, y)
```

Input: Two n-bit nonnegative integers x and y, where x ≥ 0, y ≥ 1.

Output: The quotient q and remainder r of x divided by y.



[Pre-condition: x ≥ 0 (a nonnegative integer) and y > 0 (a positive integer)]

```
q := 0;  r := x;
```

[Pre-condition Prc: x ≥ 0 and y > 0,

r = x and q = 0.]

[I(n): r = x – n y ≥ 0 and n = q.]

```
while (r ≥ y) do
   { r := r – y;
     q := q + 1};
```

// takes 2^n iterations for the worse case.

// O(n) for each r – y, where y is n bits long.

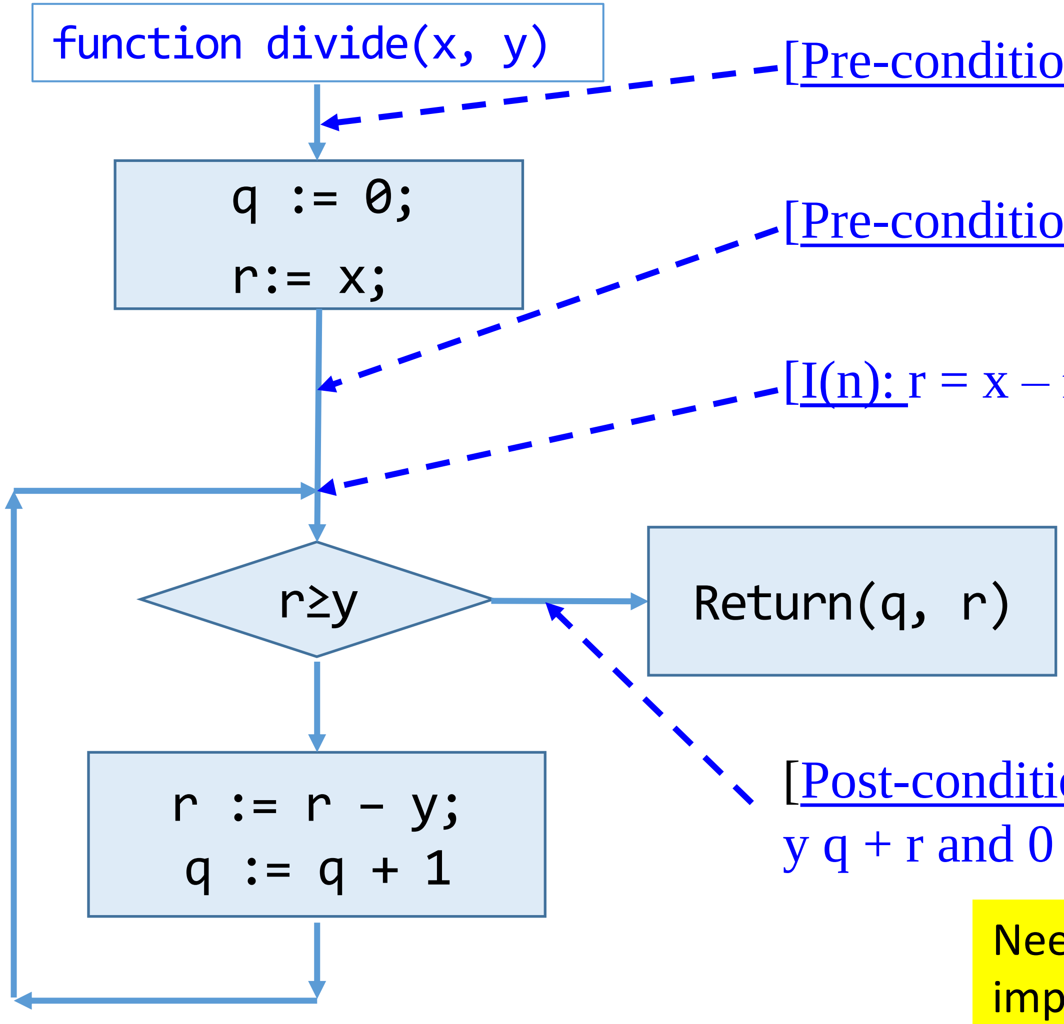
[Post-condition: q ≥ 0 and r ≥ 0 such that x = y q + r and 0 ≤ r < y.] …. Poc

```
return (q, r);
```

Need to show that Prc implies I(0) and I(k) ∩ (r ≥ y) implies I(k+1), after k+1 iteration.

Need to show that I(n) ∩ ¬(r ≥ y) implies Poc.

Correctness of the Division Algorithm



[Pre-condition: $x \geq 0$ (a nonnegative integer) and $y > 0$ (a positive integer)]

[Pre-condition Prc: $x \geq 0$ and $y > 0$, and $r = x$ and $q = 0$.]

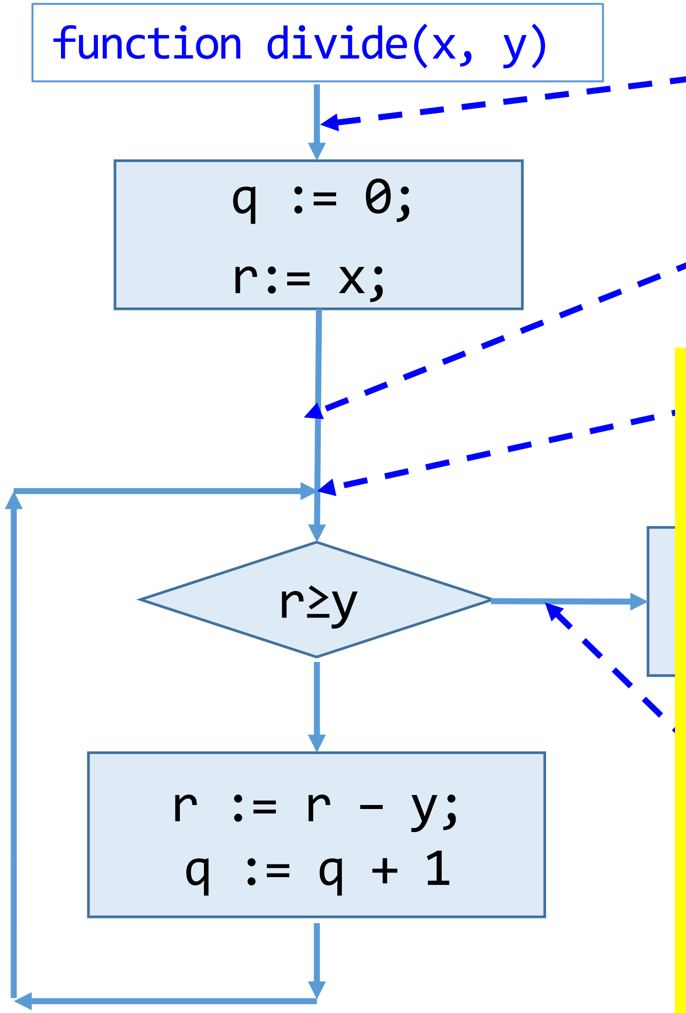
[I(n): $r = x - n\,y \geq 0$ and $n = q$.]

Need to show that Prc implies I(0) and I(k) ∩ (r ≥ y) implies I(k+1), after k+1 iteration.

[Post-condition Poc: $q \geq 0$ and $r \geq 0$ such that $x = y\,q + r$ and $0 \leq r < y$.] …. Poc

Need to show that I(n) ∩ ¬(r ≥ y) implies Poc.

Correctness of the Division Algorithm



[Pre-condition: $x \geq 0$ (a nonnegative integer) and $y > 0$ (a positive integer)]

[Pre-condition Prc: $x \geq 0$ and $y > 0$, and $r = x$ and $q = 0$.]

Prove: That Pre-condition implies Prc is true.

Proof:

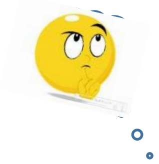
- $A \rightarrow B \equiv \neg A \cup B$. To show that "Pre-condition → Prc" is true, we need to show Prc is true if Pre-condition is true. i.e., (¬ true ∪ true) is true.
- Assume that x and y inputs are two n-bit nonnegative integers $x \geq 0$ and $y > 0$, which is the Pre-condition. Then ¬Pre-condition is false.
- Then we need to show that Prc is true, if "¬Pre-condition ∪ Pre" is true.
- Pre is true because by assumption $x \geq 0$ and $y > 0$, and $x= x$ and $0 = 0$ after executing the two assignment statements `q := 0;r:= x;` .

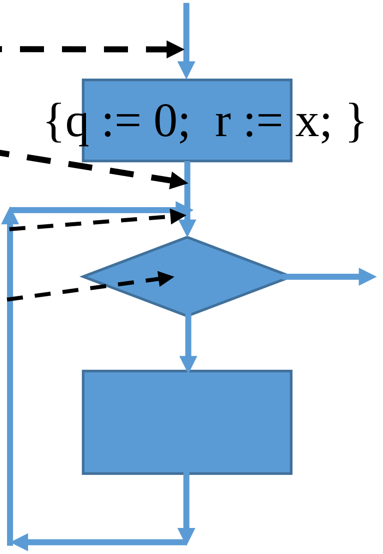
Correctness of the Division Algorithm

$A \rightarrow B \equiv \neg A \cup B$

[Pre-condition: $x \geq 0$ and $y > 0$]

Proof: [Pre-condition Prc: $x \geq 0$ and $y > 0$, $r = x$ and $q = 0$.]





To prove the correctness of the loop, let the loop invariant be

I(n): $r = x – n\,y \geq 0$ and $n = q$.

The guard of the while loop is

G: $r \geq y$

I. Basis Property: [I(0) is true before the first iteration of the loop.]

Need to show: Prc → I(0) is true. That is,

[I(0): “ $\mathbf{r = x – 0{*}y \geq 0}$ and $q = 0$”, when $n = 0$] is true.

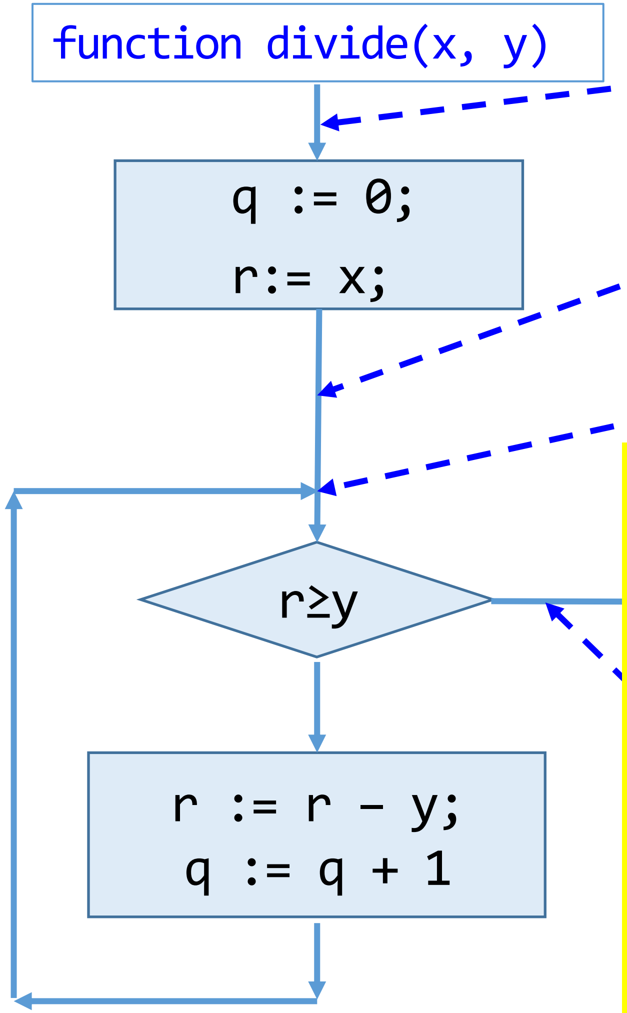
The pre-condition Prc states that $r = x$, $x \geq 0$, and $q = 0$.

Since $r = x$, $x \geq 0$ (given in Prc) and $x = x - 0{*}y$, then substitute $x = x - 0{*}y$ in $r = x$. This yields $\mathbf{r = x – 0{*}y}$. And since $x \geq 0$

I(0) : $\mathbf{r = x – 0{*}y} \geq 0$ and $0 = q$ is true before the first iteration of the loop.

II. Inductive Property: [Let n be $k \geq 0$ iterations. ...]

Correctness of the Division Algorithm



[Pre-condition: $x \geq 0$ (a nonnegative integer) and $y > 0$ (a positive integer)]

[Pre-condition Prc: $x \geq 0$ and $y > 0$, and $r = x$ and $q = 0$.]

[I(n): $r = x – n\ y \geq 0$ and $n = q$.]

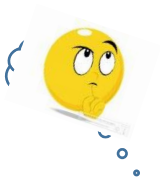
I. Basis Property:

Prove: That Prc implies I(0) is true.

Proof:

- Use $A \rightarrow B \equiv \neg A \cup B$ to prove the implication is true.
- Assume that Prc: $x \geq 0$ and $y > 0$, and $r = x$ and $q = 0$ is true.
- Need to show that "I(0): $r = x - 0*y \geq 0$ and $0 = q$". is true, before entering the loop, $n = 0$. [Note that I(0) is derived from I(n), when $n = 0$]
- By assumption $x \geq 0$, then we write $x = x – 0*y \geq 0$
- By assumption $r = x$, then $r = x – 0*y \geq 0$ by subsituting $x = x – 0*y \geq 0$ in $r = x$.
- Thus, I(0): "$r = x – 0 * y \geq 0$ and $0 = q$" is true.

Correctness of the Division Algorithm



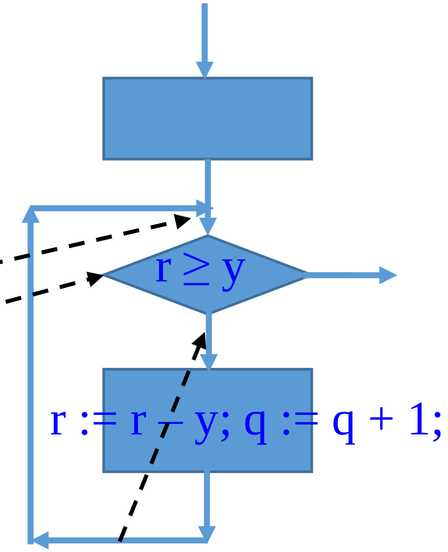
Proof:

To prove the correctness of the loop, let the loop invariant be

I(n): $\mathbf{r = x - n\,y \geq 0}$ **and** $\mathbf{n = q}$.

The guard of the while loop is

G: $r \geq y$



II. Inductive Property: [If G ∧ I(k) is true before k+1th loop's iteration (where $k \geq 0$, a nonnegative integer), then I(k+1) is true after iteration of the loop.]

G ∧ I(k), $k \geq 0$ → I(k+1)

Suppose $k \geq \mathbf{0}$ such that G ∧ I(k) is true before k+1th iteration of the loop.

Since G: $r \geq y$ is true, the loop is entered. Since I(k) is true, that is,

I(k): $r = x - k\,y \geq 0$ and $k = q$ is true.

Before execution of statements "r := r – y; q := q +1;",

G: $r_k \geq y$ and I(k): $r_k = x - k\,y \geq 0$ and $q_k = k$.

Executing these statements "r := r – y; q := q + 1;" , we obtain

$$\mathbf{r_{k+1} = r_k - y = x - k\,y - y = x - (k+1)\,y} \qquad \text{..........(D.01)}$$

and $\mathbf{q_{k+1} = q_k + 1 = k + 1.}$ ....(D.02)

G ∧ I(k) =
$(r_k \geq y) \wedge (r_k = x - k\,y \geq 0$ and $q_k = k.)$
→
~~$r_{k+1} = r_k - y \geq 0$ and~~
$\mathbf{(r_{k+1} = x - (k+1)\,y \geq 0}$ **and** $\mathbf{q = k + 1,)}$

I(n): $\mathbf{r = x - n\ y \geq 0}$ **and** $\mathbf{n = q}$.

The guard of the while loop is G: $r \geq y$

since $r_k \geq y$ before execution of statements "r := r – y; q := q + 1;",

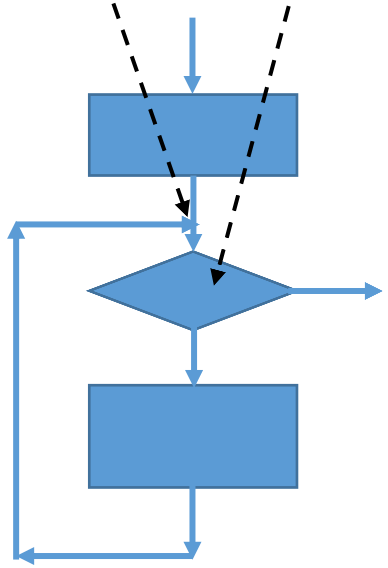
after execution of these statements

$$\mathbf{r_{k+1} = r_k - y \;\geq\; y - y \geq 0.} \qquad \ldots\ldots\ldots\ldots\ldots\ldots..(D.03)$$

Combine these equations (D.01): $\mathbf{r_{k+1} = r_k - y = x - (k+1)\ y}$,

(D.02): $\mathbf{q_{k+1} = q_k + 1 = k + 1}$, and

(D.03): $\mathbf{r_{k+1} = r_k - y \geq 0}$



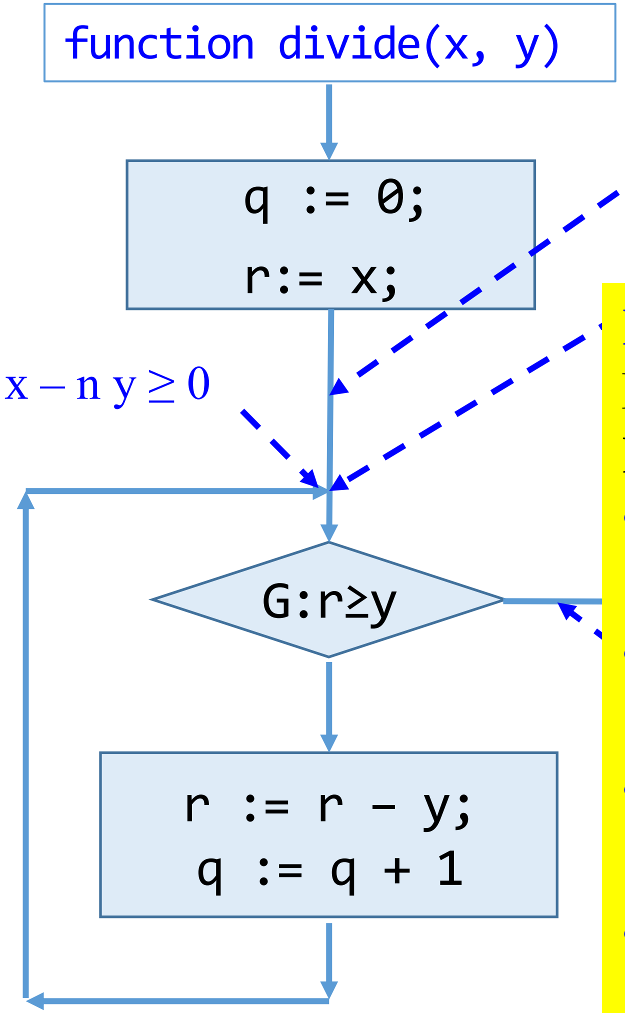
to yield that after iteration of the loops,

$\mathbf{r_{k+1} \geq 0}$ and $\mathbf{r_{k+1} = x - (k+1)\ y \geq 0}$ and $\mathbf{q = k + 1.}$

Hence I(k+1): $r_{k+1} = x - (k+1)\ y \geq 0$ and $q = k + 1$ is true.

III. Eventual Falsity of the Guard: [After a finite number of iterations of the loop, G: $r \geq y$ becomes false.]

Correctness of the Division Algorithm



[Pre-condition Prc: $x \geq 0$ and $y > 0$, and $r = x$ and $q = 0$.]

[I(n = k): $r = x - n\,y \geq 0$ and $n = q$.]

[I(n = k + 1): $r = x - n\,y \geq 0$ and $n = q$.]

II. Inductive Property:

Prove: $G \wedge I(k)$, $k \geq 0 \rightarrow I(k+1)$ is true.

Proof:

- Suppose $k \geq 0$ such that $G \wedge I(k)$ is true before $k+1^{th}$ iteration of the loop.
- Since G: $r \geq y$ is true, the loop is entered. Since I(k) is true, that is, I(k): $r = x - k\,y \geq 0$ and $k = q$ is true.
- Before execution of statements "r := r – y; q := q +1;",

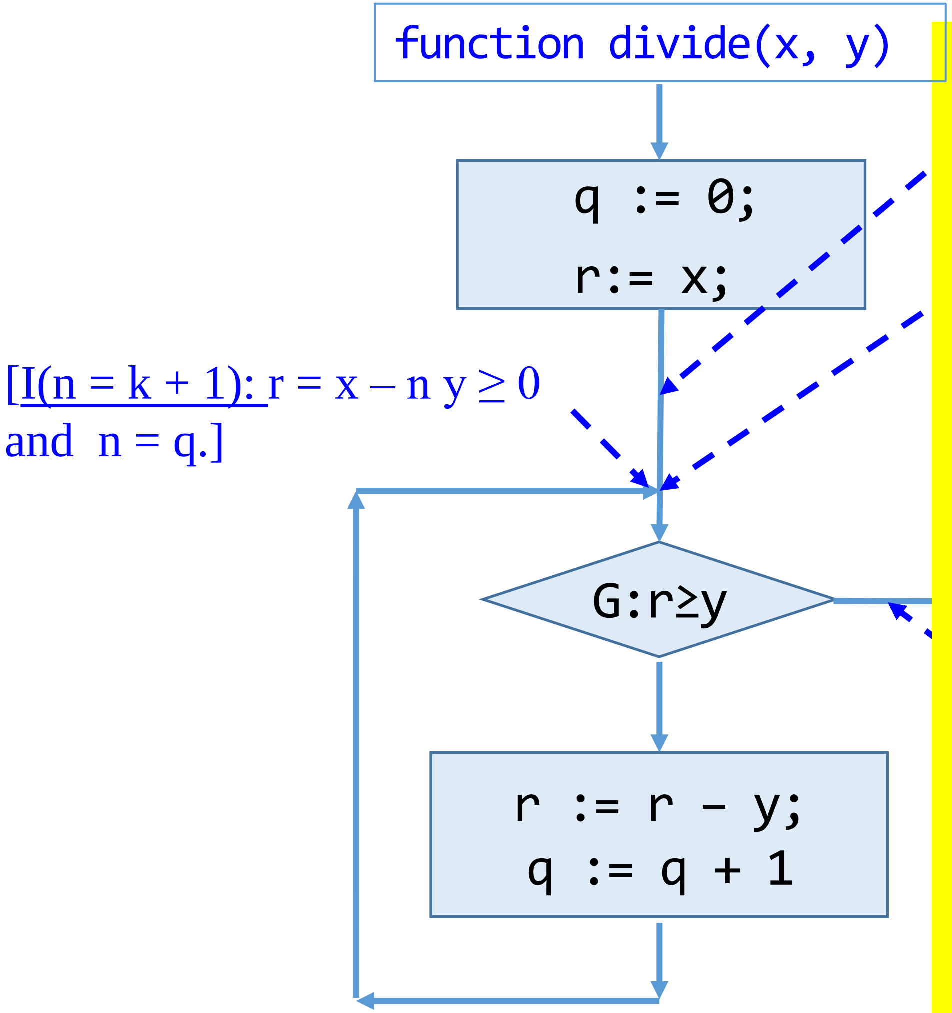
  G: $r_k \geq y$ and I(k): $r_k = x - k\,y \geq 0$ and $q_k = k$.
- Executing these statements "r := r – y; q := q + 1;" , we obtain

  $\mathbf{r_{k+1} = r_k - y = x - k\,y - y = x - (k + 1)\,y}$ ….(D.01)

  and $\mathbf{q_{k+1} = q_k + 1 = k + 1.}$ ……….(D.02)

  $\mathbf{r_{k+1} = r_k - y \geq y - y \geq 0}$, since $r_k \geq y$ ……..(D.03)

Correctness of the Division Algorithm



[Pre-condition Prc: $x > 0$ and $y > 0$, and

II. Inductive Property:

Prove: $G \wedge I(k)$, $k \geq 0 \rightarrow I(k+1)$ is true.

Proof:

- Suppose $k \geq 0$ such that $G \wedge I(k)$ is true before k+1[th] iteration of the loop.
- Since G: $r \geq y$ is true, the loop is entered. Since I(k) is true, that is, I(k): $r = x - k\, y \geq 0$ and $k = q$ is true.
- Before execution of statements "r := r – y; q := q +1;",

  G: $r_k \geq y$ and I(k): $r_k = x - k\, y \geq 0$ and $q_k = k$.

- Executing these statements "r := r – y; q := q + 1;" , we obtain

  $\mathbf{r_{k+1} = r_k - y = x - k\,y - y = x - (k+1)\,y}$ ….(D.01)

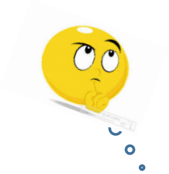
  and $\mathbf{q_{k+1} = q_k + 1 = k + 1.}$ ……….(D.02)

  $\mathbf{r_{k+1} = r_k - y \geq y - y \geq 0}$, since $r_k \geq y$ ……...(D.03)

- Combine these equations to yield that, after iteration of the loops,

  $\mathbf{r_{k+1} \geq 0}$ and $\mathbf{r_{k+1} = x - (k+1)\,y \geq 0}$ and $\mathbf{q = k + 1.}$

Hence I(k+1): $r_{k+1} = x - (k + 1)\, y \geq 0$ and $q = k + 1$ is true.



III. Eventual Falsity of the Guard: [After a finite number of iterations of the loop, the condition of G becomes false.]

```
while r ≥ y do
  { r := r – y;
    q := q + 1};
```

The Guard G is $r \geq y$. For each iteration of the loop, $r = r - y$ and $r \geq 0$, r is always a nonnegative value. The values of r form a decreasing sequence of nonnegative integers. By the well-ordering principle, there must have a smallest r, say r_{min}. Then $r_{min} < y$.

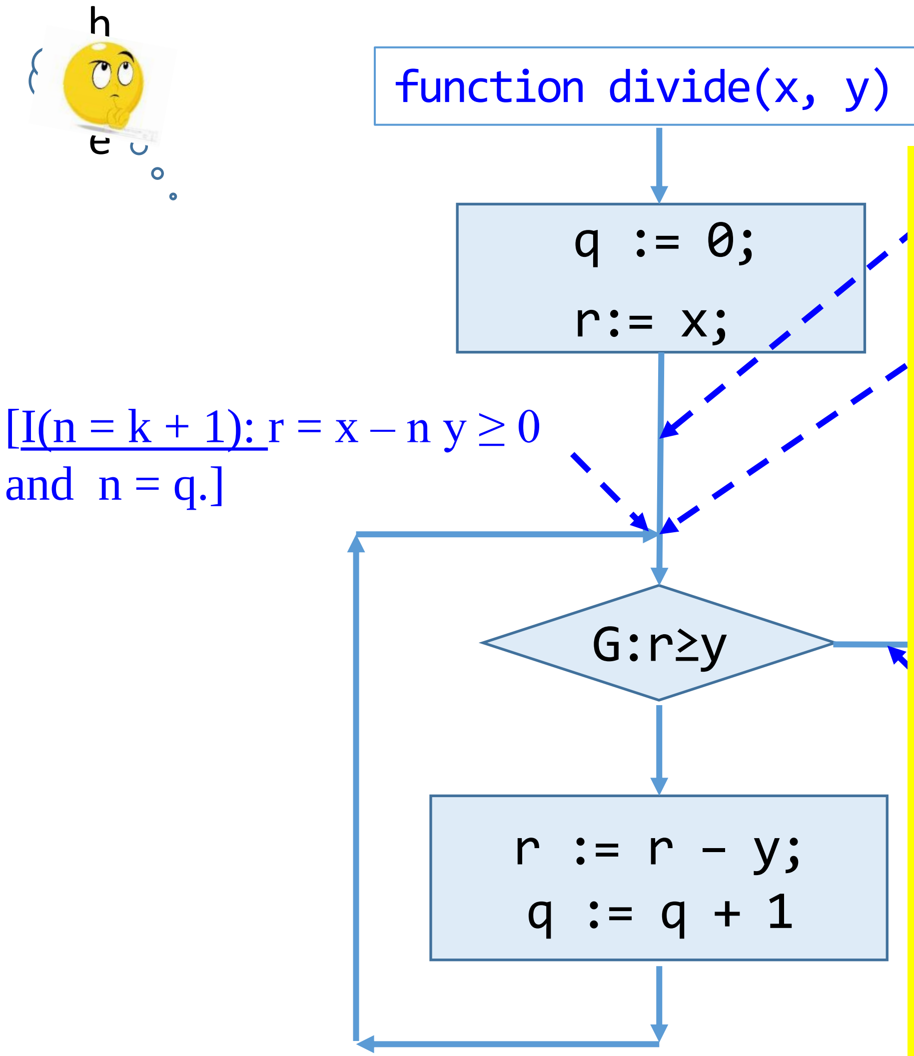
[If $r_{min} \geq y$, there will have one more time iteration of the loop and generate a new value of $r = r_{min} – y$, such that $r < r_{min}$. This would contradict the fact that r_{min} is the smallest remainder obtained by repeated iterations of the loop.]

Hence when the value $r = r_{min}$ is computed, then $r < y$. So the guard G is false.

Correctness of the Division Algorithm

[Pre-condition Prc: $x \geq 0$ and $y > 0$, and

$r = x$ and $q = 0$.]



III. Correctness of the Post-Condition

Prove: For the least number of iterations N, G: $r \geq y$ is false, and I(N) is true, then Poc is true.

Proof: The Guard G is $r \geq y$.

- For each iteration of the loop, $r = r - y$ and $r \geq 0$, r is always a nonnegative value. $r > r - y > r - 2y > \ldots > r - ny$, for $n > 0$; The values of r form a decreasing sequence of nonnegative integers.
- By the well-ordering principle, there is an n such that $r - ny > y$ and $r - (n+1)y < y$. i.e., there must have a smallest r, say r_{min}. Then $r_{min} < y$.
- [If $r_{min} \geq y$, there will have one more time iteration of the loop and generate a new value of $r = r_{min} - y$, such that $r < r_{min}$. This would contradict the fact that r_{min} is the smallest remainder obtained by repeated iterations of the loop.]
- Hence when the value $r = r_{min}$ is computed, then $r < y$. So the guard G is false

IV. Correctness of the Post-Condition: [If N is the least number of iterations, G: $r \geq y$ becomes false, and I(N) is true, then the values of the algorithm variables will be as specified in the post-condition of the loop.]

Need to show that $I(n) \cap \neg(r \geq y)$ implies Poc

[The Post-condition Poc: $q \geq 0$ and $r \geq 0$ such that $x = y\,q + r$ and $0 \leq r < y$.]

For I(n): $r = x - n\,y \geq 0$ and $n = q$, suppose that for some nonnegative integer $N \geq 0$, after N iterations,

I(N): $r = x - N*y \geq 0$ and $N = q$ is true, and G is false (i.e. $r < y$).

Since $N \geq 0$ and $N = q$, this says $q \geq 0$.

Then $r < y$, $r \geq 0$, $r = x - Ny$, and $q = N$.

Since $q = N$, by substitution, $r = x - Ny$ yields

$$r = x - q\,y.$$

That is, $x = q\,y + r$.

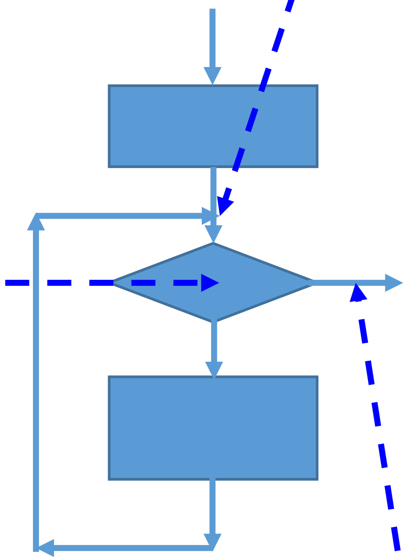
Combining the two inequalities $r < y$, $r \geq 0$ gives

$$0 \leq r < y.$$

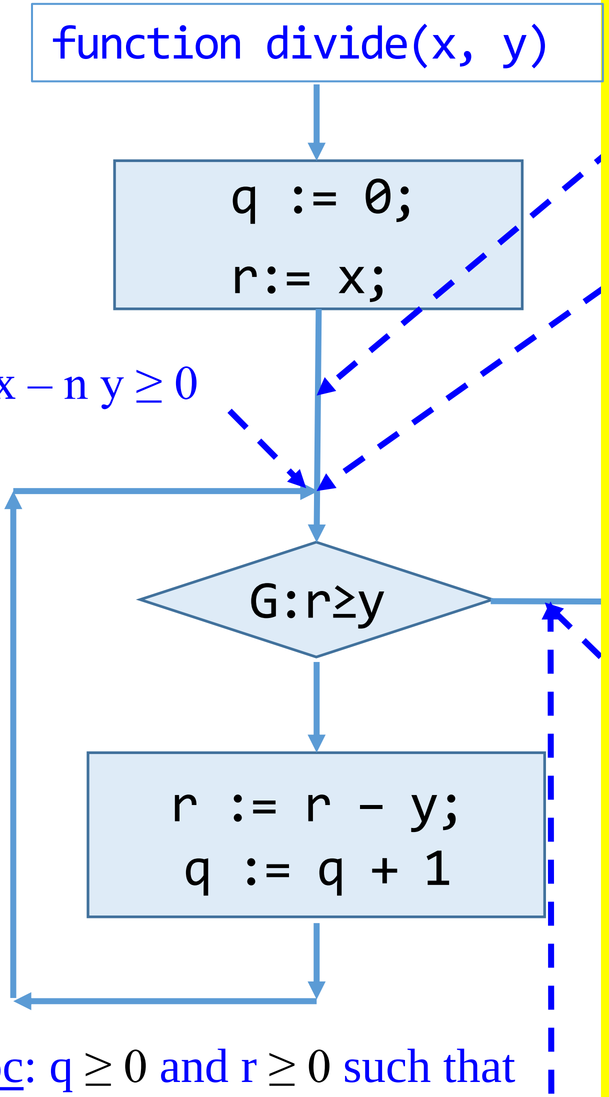
These are the values of q and r specified in the post-condition. The proof is complete.

[I(n): $r = x - n\,y \geq 0$ and $n = q$.]

G: $r \geq y$



Post-condition Poc: $q \geq 0$ and $r \geq 0$ such that $x = y\,q + r$ and $0 \leq r < y$.]



[I(n = k + 1): r = x – n y ≥ 0 and n = q.]

[Post-condition Poc: q ≥ 0 and r ≥ 0 such that x = y q + r and 0 ≤ r < y.] …. Poc

III. Correctness of the Post-Condition

Prove: For the least number of iterations N, G: r ≥ y is false, and I(N) is true, then Poc is true.

Proof:

- Suppose that for some nonnegative integer N ≥ 0, after N iterations,

  I(N): **r** = x – N*y **≥** ***0*** and N = q is true, and G is false (i.e. r < y).

- ***q* ≥ *0*,** because N ≥ 0 and N = q. Then **r < y,** ***r* ≥ *0*,** r = x – Ny, and q = N **≥** ***0***.

- Since q = N, by substitution, r = x – Ny yields

  r = x - q y.

  That is, ***x = q y + r.***

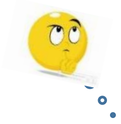
- Combining the two inequalities **r < y,** ***r* ≥ *0*** gives

  0 ≤ r < y.

  [Post-condition Poc: q ≥ 0 and r ≥ 0 such that x = y q + r and 0 ≤ r < y.] is true QED.

```
q := 0;  r := x;
while r ≥ y do
    { r := r – y ;
      q := q + 1;}
```

Suppose x and y are each n bits long. The value x and y be $2^n - 1$. Let r = x.



- The subtraction of y from r (i.e., r – y) is at most n bits.
- Compute the difference of their individual bits in a fixed amount of tim c_1. It is r := r + (-y).
- The total running time for the addition algorithm is $c_0 + c_1 n$, where c_0 and c_1 are some constants.
- Thus, the running time is O(n) for each of r := r – y. It is linear.

The algorithm yields $(x - i * y) < y$ at i iterations (i.e., q = i).

x = q*y + r
$0 \le r < y$.

- For each iteration, it takes $2(c_0 + c_1 n)$ additions/subtraction.
- Then $x < (i + 1) * y$, which is $\frac{x}{y} < i + 1$ for exiting from the iteration.

15 is 01111.
15 < 1
15 – 15*1 < 1
15 < (15+1)*1

- If x is n bits long, then x has the maximum value of x, $0 < x \le 2^n - 1 < 2^n$.
- If x >> y, the min(y) =1, then $\frac{x}{y}$ will grow to 2^n. (Both $x \ge 0$ and $y > 0$ are integers.)
- Thus, i will be approximately equal to 2^n.

Then the algorithm will take $\sum_1^{2^n} 2(c_0 + c_1 n) = 2^n * (2(c_0 + c_1 n)) = O(n\, 2^n)$, which is exponential time to execute these{ r := r – y; q := q + 1;}

Division $\frac{x}{y} = (q, r), where\ y \neq 0.$

$x = y * q + r$ and $0 \leq r < y$.

The recursive version of division in Figure 1.2 is as follows:

x = q * y + r , where r < y.
(5, 2)⇐D(17, 3)
(2, 2)⇐(q, r) := D(8, 3)
(1, 1)⇐(q, r) := D(4, 3)
(0, 2)⇐(q, r) := D(2, 3)
(0, 1)⇐(q, r) := D(1, 3)
(0, 0)⇐(q, r) := D(0, 3)

Figure 1.2 The recursive version of

```
function divide(x, y)
```

Input: Two n-bit integers x and y, where $x \geq 0$. $y \geq 1$.

Output: The quotient and remainder of x divided by y.

```
if (x = 0) then return (q, r):=(0, 0);
(q, r) := divide(⌊x/2⌋, y )    //requires n-bits right shift
q := 2 * q,  r := 2 * r;        // shift left one bit.
if (x is odd) then r := r + 1;  // needs c*n-bits
if (r ≥ y) then                 // additions
    { r := r – y; q := q + 1};
return (q, r);
```

The total time taken is thus $O(n^2)$.

x = q * y + r
D(0, 3) 0 = 0 * 3 + 0
D(1, 3) q = 2*0, r = 2*0
r = 0 + 1
1 = 0 * 3 + 1
D(2, 3) q = 2*0, r = 2*1
2 = 0*3 + 2
D(4, 3) q = 2*0, r = 2*2
(r≥y) = (4≥3)
r=4-3, q=0+1
4 = 1*3 + 1
D(8,3)q=2*1, r=2*1
8 = 2*3 + 2
D(17,3)q=2*2, r=2*2
15 is odd, r=4+1
(r≥y)=5≥3, r=5-3,
q=4+1; 17=5*3+2

Figure 1.2 The recursive version of division

```
function divide(x, y)
```

Input: Two n-bit integers x and y, where y ≥ 1.

Output: The quotient and remainder of x divided by y.

```
if (x = 0) then return (q, r):=(0, 0);
(q, r) := divide(⌊x/2⌋, y )
q := 2 * q,  r := 2 * r;
if (x is odd) then r := r + 1;
if (r ≥ y) then
{ r := r − y; q := q + 1};
return (q, r);
```

x = q * y + r , where r < y. R(q, r).
(q, r) := D(17, 3) → R(5, 2)
(q, r) := D(8, 3) → R (2, 2)
(q, r) := D(4, 3) → R(1, 1)
(q, r) := D(2, 3) → R(0, 2)
(q, r) := D(1, 3) → R(0, 1)
(q, r) := D(0, 3) → R(0, 0)

D(0, 3)
Since x = 0, **R(q=0, r=0).**

D(1, 3)
q := 2* 0, r := 2 *0
If (x =1 is odd) then r := 0 + 1=1
If (1 ≥ 3) –
R(q = 0, r =1).

D(8, 3)
q := 2* 1, r := 2 *1
If (x =8 is odd) -
If (2 ≥ 3) -
R(q = 2, r =2).

D(2, 3)
q := 2* 0, r := 2 *1
If (x =2 is odd) -
If (2 ≥ 3) –
R(q = 0, r =2).

D(4, 3)
q := 2* 0, r := 2 *2
If (x =4 is odd) -
If (4 ≥ 3) {r:= 4-3; q:= 0+1
R(q = 1, r =1).

D(17, 3)
q := 2* 2, r := 2 *2
If (x =17 is odd) then r := 4+1
If (5 ≥ 3) {r:= 5-3; q:= 4+1
R(q = 5, r =2).

Interpretation of the following statements:

```
(1) q := 2 * q,  r := 2 * r;              // shift left one bit.
(2) if (x is odd) then r := r + 1;        // needs c*n-bits
(3) if (r ≥ y) then                       // additions
        { r := r - y; q := q + 1};
return (q, r);
```

(1) $x/2 = q*y + r$

$x = 2*q*y + 2*r$ where $Q = 2*q$, $R = 2*r$

$x = \mathbf{Q}*y + \mathbf{R}$

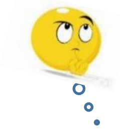
(2) when x is odd as it is flooring or rounding down the value of x the representation would be $(x - 1) / 2 = q*y + r$

$x - 1 = 2*q*y + 2*r$

$x = 2*q*y + 2*r + 1$

$x = Q*y + \mathbf{(R + 1)}$ where $Q = 2*q$, $R = 2*r$

(3) It performs the usual function of a division operation. It checks whether the value of reminder, r greater or equal to y. If true, updates the value of r to be *(r – y)* and increases Quotient q by 1.



Prove $T(n) = O(n^2)$

$T(n) = T(\lfloor n/2 \rfloor) + c(n)$

$T(1) = c_0$ (assume $c_0 = 1$);

Solution:

(need to check the correctness of the following)

Let $n = 2^k$.

$$
\begin{aligned}
T(n) = T(2^k) &= T(2^{k-1}) + 2^k \\
&= T(2^{k--2}) + 2^{k-1} + 2^k \\
&= \ldots \\
&= T(2^{k-i}) + (2^{k-i+1}) + (2^{k-i+2}) + \ldots + (2^{k-3}) + (2^{k-2}) + (2^{k-1}) + (2^k) \\
&= T(2^{k-k}) + (2^{k-k+1}) + (2^{k-k+2}) + \ldots + (2^{k-3}) + (2^{k-2}) + (2^{k-1}) + (2^k),\ k = i \\
&= \color{red}{T(2^{k-k})} + (2^{k-k+1}) + (2^{k-k+2}) + \ldots + (2^{k-3}) + (2^{k-2}) + (2^{k-1}) + (2^k), \\
&= 1 + (2^1) + (2^2) + \ldots + (2^{k-3}) + (2^{k-2}) + (2^{k-1}) + (2^k), \\
&= (2^{k+1} - 1) \\
&= 2n - 1 = O(n) \text{ recursive calls}
\end{aligned}
$$

```
function divide(x, y)
    if  x = 0, then return (q, r) := (0, 0);
    (q, r) := divide( ⌊ x/2 ⌋ , y )
    q := 2 * q,  r := 2 * r;
    if x is odd then r := r +1;
    if r ≥ y then { r := r – y; q := q +1};
    return (q, r)
```

The algorithm will take n calls, and therefore $O(n^2)$.

End of Proof of Program Correction.