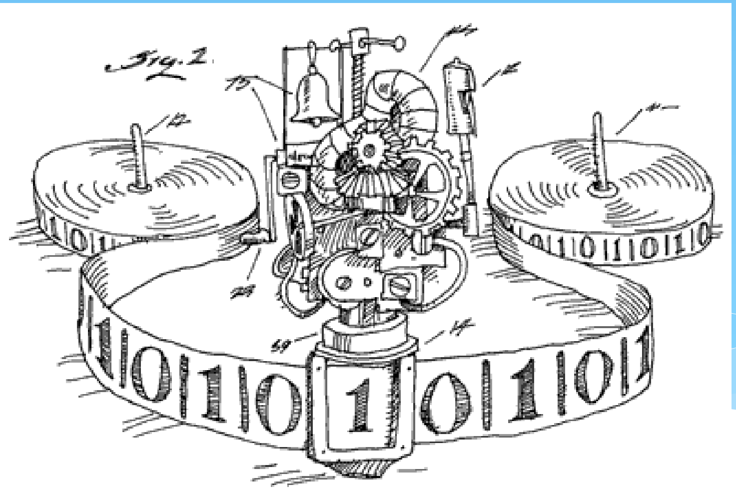
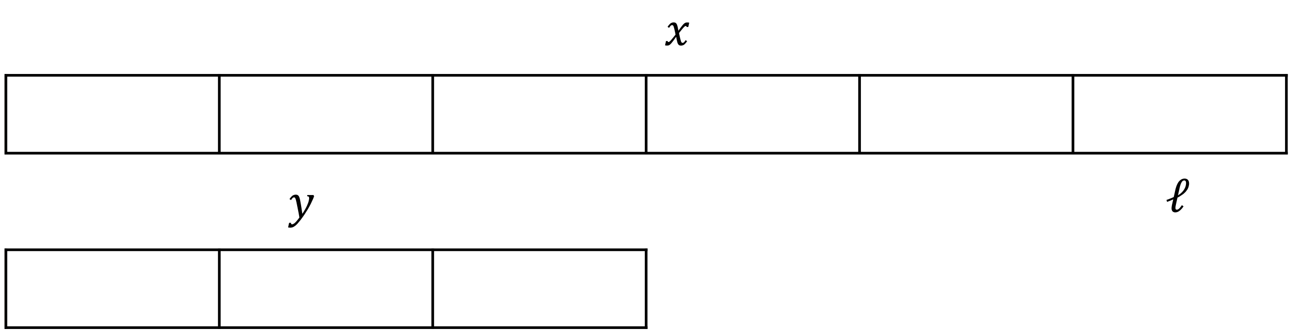
EECS 376: Foundations of Computer Science

Euiwoong Lee



Coding Interview Question
(The Linear Tiling Problem)

* We have two dirt paths of integer lengths $x \geq y > 0$
* We want to make them nice sidewalks by laying down cement blocks of the same integer length ℓ such that the blocks tile both paths
* **Goal:** Find length ℓ that *minimizes* # of blocks



Step 1: Give naïve solution

- **Tip:** Start with the easiest solution that works.
- **Naïve solution:** Try various lengths ℓ.
  - **Q:** In what order should we try ℓ?
    - Work our way down from $y, y-1, \ldots, 2,1$
  - **Q:** What does it mean to "try" a tile length?
    - If ℓ *divides* x **and** y, then return ℓ.
- **Interviewer:** "Why is this a bad solution?"

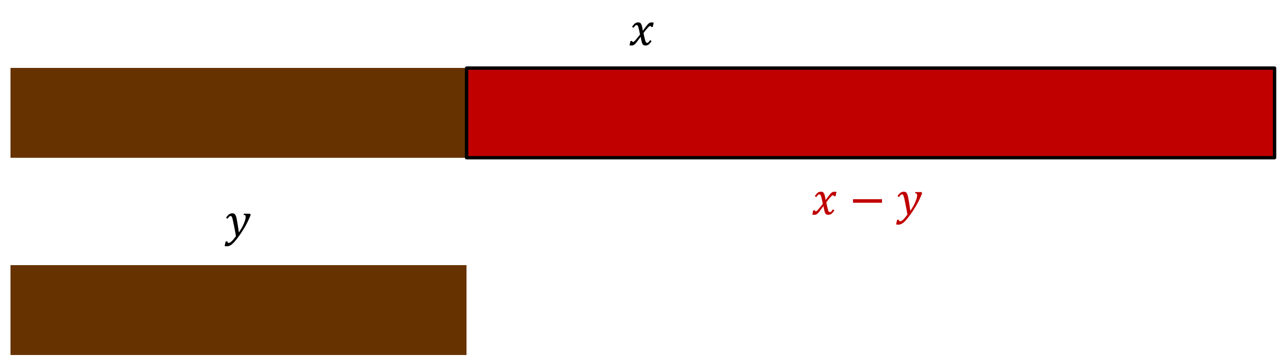
Step 2: Analyze runtime of the naïve solution

- **Q:** Suppose x and y each have n digits. How large can y be?
  - $10^n - 1$
- **Q:** What's the runtime of the naïve algorithm? **Recall:** "size" of an integer is $O(\#\text{ digits})$
  - *Exponential in the size of the input! (Not efficient.)*

Naïve(x, y):
for $\ell = y, y-1, \ldots, 1$:
  if ℓ divides x and y,
    return ℓ

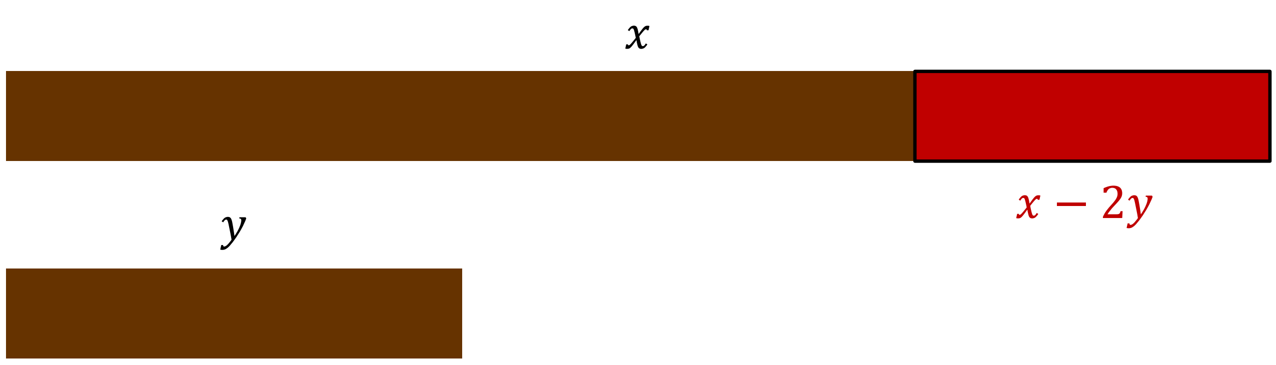
Step 3: Think strategic

- **Tip:** It's often fruitful to try to simplify the problem (ideally, into one that you know how to solve).
- **Strategy:** Recursively solve the problem
- **Interviewer:** "Could we work on lengths $x - y$ and y instead of x and y? Is that equivalent?"



How far can we reduce?

- In general, we can reduce k times until $x - ky < y$.
- **Q:** How large can $x - ky$ be?
- **Q:** How small can $x - ky$ be?
- **Q:** What is $x - ky$? **Hint:** Think division.
  - $x \bmod y =$ the remainder of x divided by y



Step 4: Code it up

- We have just discovered the ***Euclidean Algorithm***
- Euclid invented in ≈ 300 BC it to compute the ***greatest common divisor*** of two integers
- **Interviewer:** "What is the runtime of Euclid?"
- This is a tricky question! We need some tools…

Euclid(x, y): // for $x > y \geq 0$
if $y = 0$: return x
if $y = 1$: return 1
return **Euclid**$(y, x \bmod y)$

Euclid, 300 BCE

Flipping game: Michigan vs Ohio State

- 11 × 11 board covered with two-sided chips:
- Two players: "row" player R and "column" player C
- **Rules:** R can flip a row/C can flip a column that has more than
- If no such move is possible, the player loses the game.
- **Question:** will the game always end? (or can the game go on forever?)

Flipping game: Michigan vs Ohio State

- 11×11 board covered with two-sided chips:
- Two players: "row" player R and "column" player C
- **Rules:** R can flip a row/C can flip a column that has more than
- If no such move is possible, the player loses the game.
- **Question:** will the game always end? (or can the game go on forever?)
- **Observation:** each row/column flip decreases the number of
- **Conclusion:** the game will always end after at most 121 steps.

EECS ECE CSE

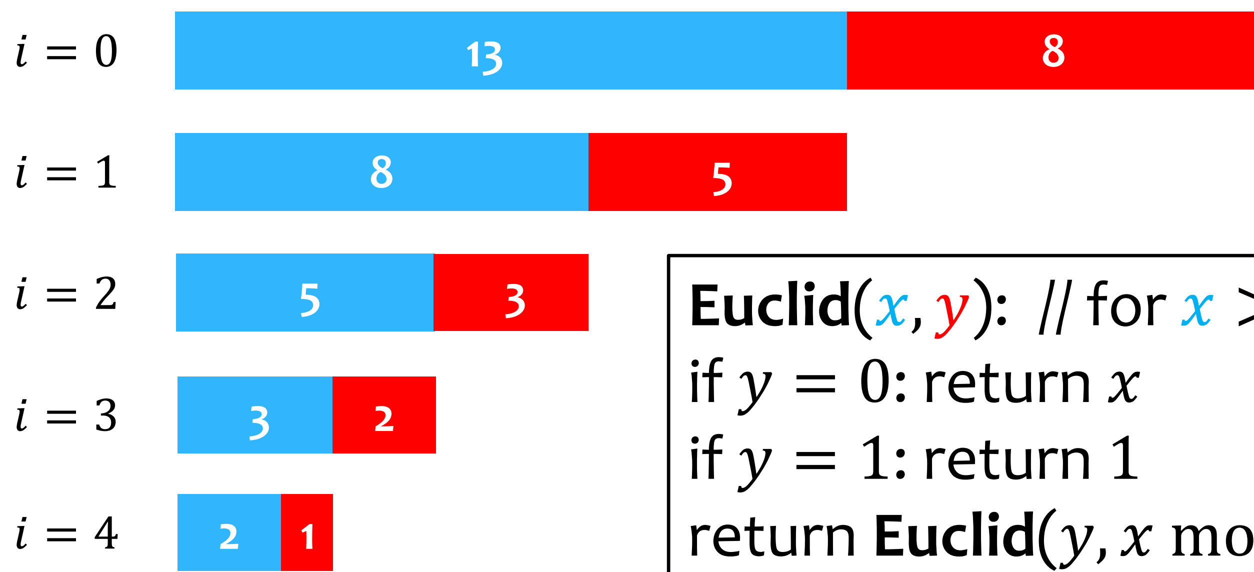
Potential Function Arguments

* A ***potential function argument*** exploits the following intuitive fact: If I start with a *finite* amount of water in a *leaky* bucket, then *eventually* the water stops leaking out
* Given some "process" (e.g., execution of an algorithm) that we wish to show terminates, a ***potential function*** defines an *integer* quantity (amount of water) that *decreases* in each "time step" of the process (leaking) and *bounded below* (can't leak forever)
* **Example:** distance to destination, loop counter, number of Ohio state chips on the board, argument in call

Observation: If we can define a potential function for a process, then *it must eventually terminate*.

In search of a potential

- **Q:** What *decreases* in a recursive call to Euclid?
  - 1st arg ($x \neq y$) and 2nd arg ($x \bmod y < y$)



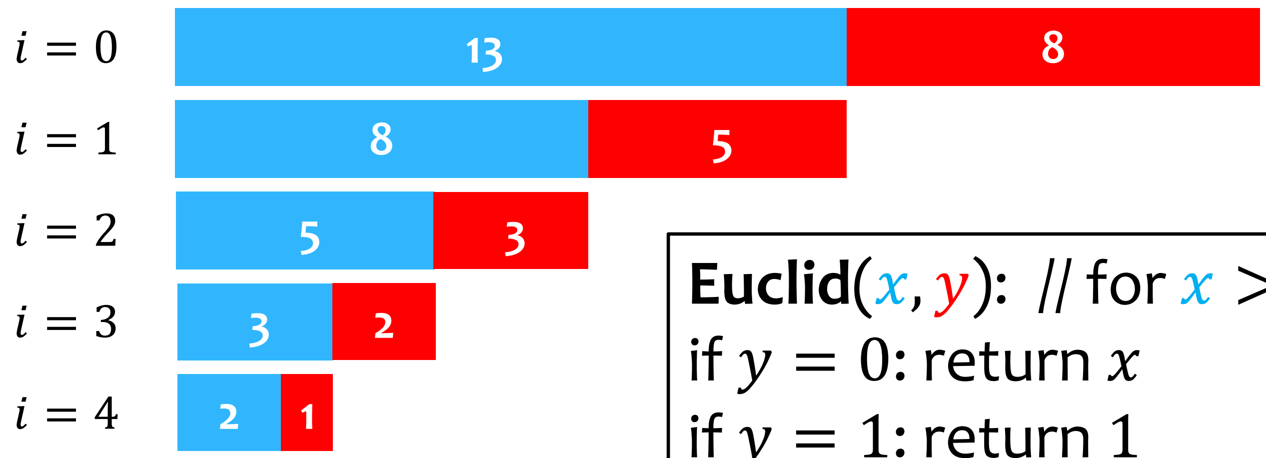
Euclid(x, y): // for $x > y \geq 0$
if $y = 0$: return x
if $y = 1$: return 1
return **Euclid**$(y, x \bmod y)$

The Euclidean algorithm terminates

* Two potential function choices:
  * 1st arg and 2nd arg both work
* For $i \geq 0$, let s_i be the 2nd arg of the i'th **Euclid** call.
* **Claim:** $s_{i+1} < s_i$ *unless the i'th call does not recurse.*
* By the same argument as the flipping game, **Euclid** terminates.
* **Claim: Euclid**(x, y) terminates after $\leq y$ calls.
  * No better than the naïve algorithm…?

A better potential

- **Claim:** The <u>sum</u> of the arguments $(x + y)$ is decreasing by at least $1/4$ in each recursive call!
- **Result:** The algorithm terminates after $O(\log{(x+y)})$ calls!

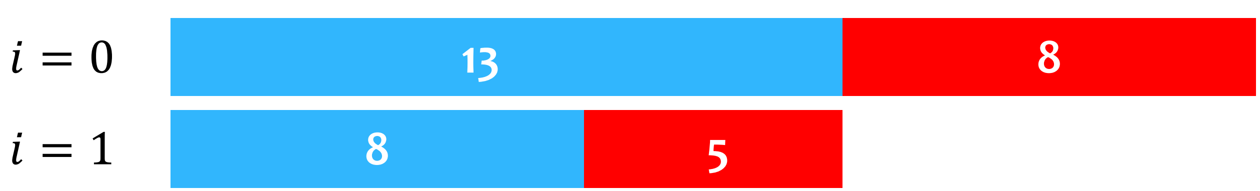


Euclid(x, y): // for $x > y \geq 0$
if $y = 0$: return x
if $y = 1$: return 1
return **Euclid**$(y, x \bmod y)$

Euclid Analysis

Euclid(x, y): // for $x > y \geq 0$
if $y = 0$: return x
if $y = 1$: return 1
return **Euclid**$(y, x \bmod y)$

- **Definitions:**
  - x_i = value of first argument after i iterations
  - y_i = value of second argument after i iterations
  - $s_i = x_i + y_i$ = potential after i iterations
- **Observations:**
  - $x_i \geq \frac{1}{2} s_i$: since $x_i > y_i$ (by design), it contributes more than half the potential
  - $x_{i+1} = y_i$
  - $y_{i+1} = x_i \bmod y_i$

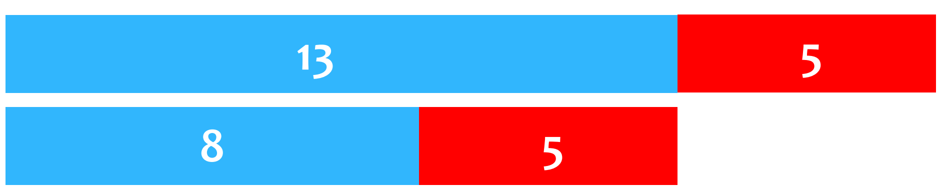


Euclid Analysis

Euclid(x, y): // for $x > y \geq 0$
if $y = 0$: return x
if $y = 1$: return 1
return **Euclid**$(y, x \bmod y)$

* **Observations:** $x_{i+1} = y_i$ and $y_{i+1} = x_i \bmod y_i$ and $x_i \geq \frac{1}{2} s_i$
* **Claim:** $y_{i+1} \leq \frac{1}{2} x_i$
  * **Case 1:** $y_i \leq \frac{1}{2} x_i$
    * Then $x_i \bmod y_i < y_i \leq \frac{1}{2} x_i$.
  * **Case 2:** $y_i > \frac{1}{2} x_i$
    * Then $x_i \bmod y_i = x_i - y_i \leq x_i - \frac{1}{2} x_i = \frac{1}{2} x_i$.

Case 1
Case 2



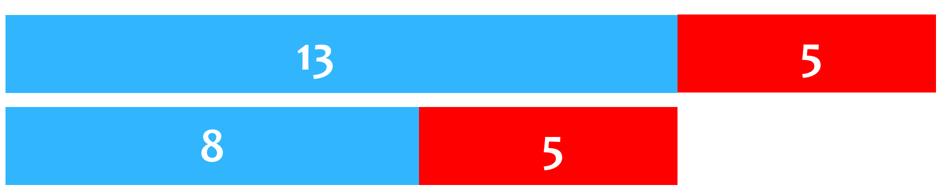
Euclid Analysis

Euclid(x, y): // for $x > y \geq 0$
if $y = 0$: return x
if $y = 1$: return 1
return **Euclid**$(y, x \bmod y)$

- **Observations:** $x_{i+1} = y_i$ and $y_{i+1} = x_i \bmod y_i$ and $x_i \geq \frac{1}{2} s_i$
- **Claim:** $y_{i+1} \leq \frac{1}{2} x_i$
- **Then:** $s_{i+1} = x_{i+1} + y_{i+1} \leq y_i + \frac{1}{2} x_i = s_i - \frac{1}{2} x_i$
  - Since $x_i \geq \frac{1}{2} s_i \Rightarrow \frac{1}{2} x_i \geq \frac{1}{4} s_i$, we're subtracting off at least $1/4$ the value of s_i, so we have at most $3/4$ left:

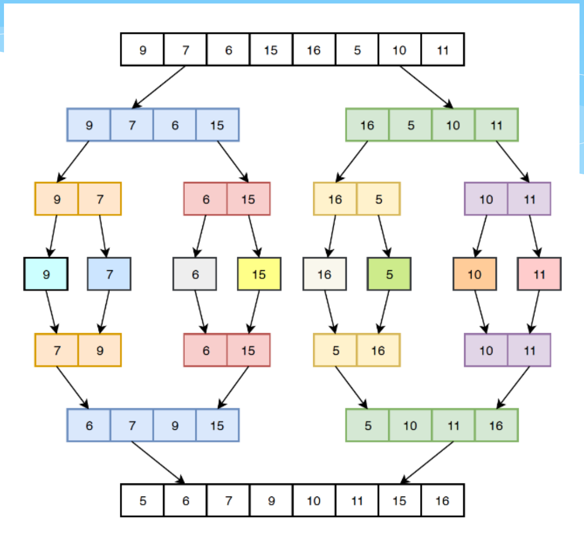
$$s_{i+1} \leq s_i - \frac{1}{2} x_i \leq \frac{3}{4} s_i$$

Case 1
Case 2



“Divide et impera” – Philip II

Algorithmic Strategy: Divide and Conquer



Divide and Conquer Algorithms

Main Idea:

1. Divide the problem into smaller subproblems
2. Solve each subproblem recursively
3. Combine the solutions of the subproblems in a "meaningful" way

Runtime Analysis:

* Tools to solve recurrence relations
* The "Master Theorem"

MergeSort

Algorithm: *MergeSort*($A[1..n]$: array of n integers)

if $n = 1$ **return**

$m \coloneqq \lfloor n/2 \rfloor$ — find mid point

MergeSort($A[1..m]$) — sort first half recursively

MergeSort($A[m+1..n]$) — sort second half recursively

return merge($A[1..m]$, $A[m+1..n]$) — combine two **sorted** lists

https://www.youtube.com/watch?v=ZRPoEKHXTJg

6	14	12	1	9	4	8	0	5	13	15	10	7	2	3	11

Sort each half recursively

0	1	4	6	8	9	12	14	2	3	5	7	10	11	13	15

Merge

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Combining two *sorted* lists

MergeSort(A[1..n]): // sorts a list of integers
if n = 1 then return A // base case
L = **MergeSort**(A[1..n/2]) // recursively sort 1^{st} half
R = **MergeSort**(A[n/2+1..n]) // recursively sort 2^{nd} half
return **Merge**(L, R) // combine solutions

- The heart of the **MergeSort** procedure is how we **Merge** the two *sorted* sublists, L and R
- **Idea:** repeatedly compare the front of L and R; pop off the smaller one and append it to the merged list

L

3	6

R

2	4	5

2	3	4	5	6

MergeSort

Algorithm: *MergeSort*($A[1..n]$: array of n integers)

if $n = 1$ **return**

$m := \lfloor n/2 \rfloor$ find mid point

MergeSort($A[1..m]$) sort first half recursively

MergeSort($A[m+1..n]$) sort second half recursively

return merge($A[1..m], A[m+1..n]$) combine two **sorted** lists

Runtime Analysis:

* $T(n)$ = runtime of MergeSort on inputs of size n.
* Runtime of combining two **sorted** arrays of size $n/2$ is $O(n)$.
* **Then:** $T(n) = 2T(n/2) + O(n)$

Question: How do we compute $T(n)$ explicitly?

The Master Theorem

Story: Divide-and-conquer algorithm breaks a problem of size n into:

* k smaller problems
* each one of size n/b
* with cost of $O(n^d)$ to combine the results together

Formally: Consider the recurrence relation $T(n) = kT(n/b) + O(n^d)$, when $k, b > 1$. Then:

$$T(n) = \begin{cases} O(n^d) & \text{if } \left(k/b^d\right) < 1 \\ O(n^d \log n) & \text{if } \left(k/b^d\right) = 1 \\ O(n^{\log_b k}) & \text{if } \left(k/b^d\right) > 1 \end{cases}$$

Back to MergeSort

Algorithm: *MergeSort*$(A[1..n]$: array of n integers)

if $n = 1$ **return**

$m := \lfloor n/2 \rfloor$ find mid point

MergeSort$(A[1..m])$ sort first half recursively

MergeSort$(A[m+1..n])$ sort second half recursively

return merge$(A[1..m], A[m+1..n])$ combine two **sorted** lists

Naïve sorting algorithms take $O(n^2)$!

Runtime Analysis:

* **Fact:** Two **sorted** arrays of size n can be combined in time $O(n)$.
* **Therefore:** $T(n) = 2T(n/2) + O(n)$.
* So $k = 2, b = 2, d = 1$.
  Therefore $k/b^d = 1$.
* **Conclusion:** $T(n) = O(n \log n)$.

$$T(n) = \begin{cases} O(n^d) & \text{if } (k/b^d) < 1 \\ O(n^d \log n) & \text{if } (k/b^d) = 1 \\ O(n^{\log_b k}) & \text{if } (k/b^d) > 1 \end{cases}$$

Integer Arithmetic

- Many programming languages support "big" integers with a *non-constant* number of digits and basic arithmetic operations on them, e.g., $+, -, *, /, \ll$, etc
  - Roughly, think of each integer as an array of digits
- How does the running time of arithmetic operations scale with the input size ($n = \#$ digits)?
  - **Addition/Subtraction:** $O(n)$
  - **Multiplication:** $O(n \log n)$ [**Harvey and Hoeven 2019**]

Integer Addition

- Given n-digit integers x and y
- **Goal:** compute $x + y$ and $x - y$
- **Easy:** add digits one at a time and keep a "carry" digit
- **Q:** What's the runtime?
  - $O(n)$

	1	1	1	
		9	4	6
+		9	8	5
	1	9	3	1

Integer Shift

- Given n-digit integer x and "small" *positive* integer k
- **Goal:** compute $x \ll k := x \cdot 10^k$ and $x \gg k := \lfloor x \cdot 10^{-k} \rfloor$
- **Easy:** "shift" the array forward or backward by k positions
- **Q:** What's the runtime?
  - $O(n + k)$

$$\mathbf{3} \quad \mathbf{7} \quad \mathbf{6} \quad \ll \quad \mathbf{2} \quad = \quad \mathbf{3} \quad \mathbf{7} \quad \mathbf{6} \quad \mathbf{0} \quad \mathbf{0}$$

$$\mathbf{3} \quad \mathbf{7} \quad \mathbf{6} \quad \mathbf{0} \quad \mathbf{0} \quad \gg \quad \mathbf{2} \quad = \quad \mathbf{3} \quad \mathbf{7} \quad \mathbf{6}$$

Integer Multiplication

- Given n-digit *positive* integers x and y
- **Goal:** compute $x * y$
- **Easy:** do "grade-school" method
- **Q:** What's the runtime?
  - $O(n^2)$ (yikes)

NaiveMult(x, y):
$r = 0$
for $i = 1..n$:
  $r \mathrel{+}= (x \cdot y[i]) \ll (i-1)$
return r

Shorthand for:
$x + x + \cdots + x$
($y[i]$ times)

		3	4
*		3	9
	3	0	6
1	0	**2**	
1	3	**2**	6

Divide and Conquer?

- **Input:** N_1 and N_2, two n-digit numbers (assume n is a power of 2)
- Split N_1 and N_2 into $n/2$ low-order & $n/2$ high-order digits:
  - $N_1 = a \cdot 10^{n/2} + b$
  - $N_2 = c \cdot 10^{n/2} + d$

	←$n/2$ digits→	←$n/2$ digits→
N_1	a	b
N_2	c	d

- Compute $N_1 \times N_2 = a \times c \cdot 10^n + (a \times d + b \times c) \cdot 10^{n/2} + b \times d$

- **Question:** Is this better than the naïve algorithm?
- **Answer:** We'll see next time!