15-251: Great Theoretical Ideas in Computer Science
Fall 2018, Lecture 9

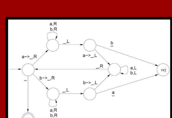
Computational Arithmetic

Today

- Talk more about intrinsic time complexity of basic problems.
- Explain a more realistic model for analyzing running time.
- Investigate these issues in the context of some very simple arithmetic problems.

Running time for PALINDROMES?

Turing Machine



T(n) = $\Theta(n^2)$

C-like pseudocode

```
lo ← 1; hi ← n
while (lo < hi)
  if x[lo] ≠ x[hi] then REJECT
  lo ← lo + 1; hi ← hi – 1
ACCEPT
```

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

reverse of string *x*

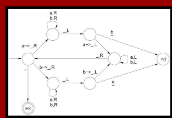
Python

```
return (x == x[::-1])
```

T(n) = $\Theta(1)$??

Which is the ri~~ght model~~?

Turing Machine



$T(n) = \Theta(n^2)$

C-like pseudocode

```
lo ← 1; hi ← n
while (lo < hi)
  if x[lo] ≠ x[hi] then REJECT
  lo ← lo + 1; hi ← hi – 1
ACCEPT
```

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

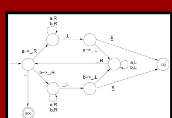
Python

```
return (x == x[::-1])
```

$T(n) = \Theta(1)$??

Which is the best model?

Turing Machine



$T(n) = \Theta(n^2)$

C-like pseudocode

```
lo ← 1; hi ← n
while (lo < hi)
  if x[lo] ≠ x[hi] then REJECT
  lo ← lo + 1; hi ← hi – 1
ACCEPT
```

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

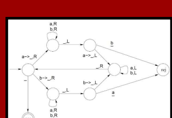
Python

```
return (x == x[::-1])
```

$T(n) = \Theta(1)$??

Which is the best model?

Turing Machine



$T(n) = \Theta(n^2)$

Reason it's quadratic: lack of "RAM" (random-access memory).

Not very realistic.

Which is the best model?

Python

```
return (x == x[::-1])
```

T(n) = Θ(1)??

Which is the best model?

Python

```
x_rev = x[::-1]
is_palindrome = (x == x_rev)
return is_palindrome
```

T(n) = Θ(1)??

Is it really "fair" to have a 1-instruction string-reverser?

Maybe just solve any language L with

```
import solveStuff
solveStuff.solveL(x)
```

?

Which is the best model?

Python

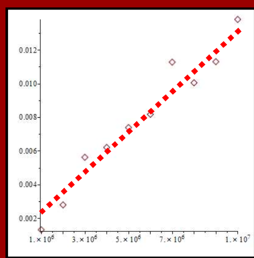
```
x_rev = x[::-1]
is_palindrome = (x == x_rev)
return is_palindrome
```

T(n) = Θ(1)??

Not even a question of fairness…

Time to do *x_rev* = *x*[::-1] on your laptop,
$n = c \cdot 10^6$, $c = 1 \ldots 10$.

seconds



n

Looks like Θ(n)

Similarly for doing *x* == *x_rev*

Why does…

```
x_rev = x[::-1]
is_palindrome = (x == x_rev)
return is_palindrome
```

…seem to take Θ(n) time?

Because it's ultimately implemented on your machine basically like this…

```
for i from 1 to n
  x_rev[i] ← x[n−i]
for i from 1 to n
  if x[i] ≠ x_rev[i] then REJECT
ACCEPT
```

RAM model

- "C-like / low-level pseudocode"
- A good, basic model for algorithmic analysis
- PROS: reasonably realistic, relatively simple, used by professional algorithmicists
- CONS: it's not easy to define it *very* precisely
- In 251, we'll try not to get hung up on precise details; they typically only make polylogarithmic-factor differences anyway ☺

RAM model

IsPalindrome(x) // $x \in \Sigma^*$ is an array of characters
$lo \leftarrow 1$
$hi \leftarrow n$ // n is the length of x
loop while ($lo < hi$)
if $x[lo] \neq x[hi]$ then REJECT
$lo \leftarrow lo + 1$
$hi \leftarrow hi - 1$
end loop
ACCEPT

Inputs are **always** strings over an alphabet Σ.
Think of Σ as like a data type, “char”.
Strings are represented as arrays of chars.

RAM model

IsPalindrome(x) // $x \in \Sigma^*$ is an array of characters
$lo \leftarrow 1$
$hi \leftarrow n$ // n is the length of x
loop while ($lo < hi$)
if $x[lo] \neq x[hi]$ then REJECT
$lo \leftarrow lo + 1$
$hi \leftarrow hi - 1$
end loop
ACCEPT

You may assume the code knows input length, n.

RAM model

IsPalindrome(x) // $x \in \Sigma^*$ is an array of characters
$lo \leftarrow 1$
$hi \leftarrow n$ // n is the length of x
loop while ($lo < hi$)
if $x[lo] \neq x[hi]$ then REJECT
$lo \leftarrow lo + 1$
$hi \leftarrow hi - 1$
end loop
ACCEPT

Big difference from TMs:
May access any memory cell in 1 step.
Can also “allocate” memory (arrays) in 1 step.

RAM model

```
IsPalindrome(x)            // x ∈ Σ* is an array of characters
  lo ← 1
  hi ← n                    // n is the length of x
  loop while (lo < hi)
    if x[lo] ≠ x[hi] then REJECT
    lo ← lo + 1
    hi ← hi – 1
  end loop
  ACCEPT
```

Basic flow-control things
  (ifs, loops, returns, assigning to variables)
count as 1 step.

RAM model

```
IsPalindrome(x)            // x ∈ Σ* is an array of characters
  lo ← 1
  hi ← n                    // n is the length of x
  loop while (lo < hi)
    if x[lo] ≠ x[hi] then REJECT
    lo ← lo + 1
    hi ← hi – 1
  end loop
  ACCEPT
```

Now it gets subtle.
We kind of want to count this a “1 step”.
But…

Can we add two integers in 1 step?

Arithmetica

Boethius

Pythagorus

Allegory of Arithmetic by Gregor Reisch, 1503.

Can we add two integers in 1 step?

"Sure. Everyone's computer has an x86 chip with an instruction for adding two registers."

"Yeah, but if the numbers have 1,000,000 binary digits, they won't fit into a 64-bit register."

Can we add two integers in 1 step?

"Million-digit numbers?! Those variables *lo* and *hi* were between 1 and *n*. Any real-world input to PALINDROMES will have $n \leq 2^{64}$ (= 10000 petabytes)."

"Yeah, but if the numbers have 1,000,000 binary digits, they won't fit into a 64-bit register."

Can we add two integers in 1 step?

"Million-digit numbers?! Those variables *lo* and *hi* were between 1 and *n*. Any real-world input to PALINDROMES will have $n \leq 2^{64}$ (= 10000 petabytes)."

"Hey, in July 2018 they found the largest known prime, $M = 2^{77,232,917} - 1$. That's 77 million binary digits. You can't store that in a register!"

Can we add two integers in 1 step?

"Million-digit numbers?! Those variables *lo* and *hi* were between 1 and *n*. Any real-world input to PALINDROMES will have $n \le 2^{64}$ (= 10000 petabytes)."

"The only way to store $M = 2^{77,232,917} - 1$ is as a *string*. Now surely doing "$M \leftarrow M + 1$" must cost something like 77 million steps."

Can we add two integers in 1 step?

"C'mon. ಠ_ಠ. Be reasonable.

```
i ← 1
while i ≤ n
    i ← i+1
```

has to be Θ(n) time."

"The only way to store $M = 2^{77,232,917} - 1$ is as a *string*. Now surely doing "$M \leftarrow M + 1$" must cost something like 77 million steps."

"Gentlemen, gentlemen, you are both right!"

"When the input length is *n*,
it is reasonable that an integer variable of *value at most n* (or n^2 or n^3) can fit in one, or a couple of registers.

We will call such a variable a

BoundedInt.

It is fair to count arithmetic operations on BoundedInts as taking O(1) steps."

"However! If the input is a number *M* which is *n* bits long (so the *value* of *M* is $\approx 2^n$), then *M* is NOT going to fit in a register.

We call such a number a

BigInt

and it must be stored as a *string*.

Any arithmetic operations on BigInts must be carried out by string-manipulation algorithms!"

RAM model: the final rules

Besides the "character" data type Σ, you may declare variables to be of type BoundedInt.

But to do this, you must separately prove that their value is $O(n^c)$ throughout the algorithm (i.e., at most polynomial in the input length).

(If this is super-duper-obvious, as in

```
i ← 1    // i is a BoundedInt
while i ≤ n
    i ← i+1
```

it's probably okay to not mention it.)

RAM model: the final rules

You can do integer arithmetic ops on BoundedInts like +, −, ×, integer-division, mod in “1 step”.

Also, whenever you access an array/memory cell

$v \leftarrow x[i]$

the index variable *i* should be a BoundedInt.

Array/memory cells can hold Σ-characters or BoundedInts, and you can convert between them in “1 step”.

RAM model

```
IsPalindrome(x)          // x ∈ Σ* is an array of characters
  lo ← 1
  hi ← n                 // n is the length of x
  loop while (lo < hi)
     if x[lo] ≠ x[hi] then REJECT
     lo ← lo + 1
     hi ← hi – 1
  end loop
  ACCEPT
```

lo and *hi* are BoundedInts.

Thus the three lines in the loop are all O(1).

So the running time is O(n). Hooray!

How long does it take to add two ~~in~~

How long does it take to add two BigInts?

How do we even do it by any algorithm??

Add(x, y)   /* x, y are BigInts, stored as strings over {0,1,2,3,4,5,6,7,8,9} of at most n digits each */

return ~~x~~

How long does it take to add two BigInts?

How do we even do it by any algorithm??

Add(x, y)   /* x, y are BigInts, stored as strings over {0,1,2,3,4,5,6,7,8,9} of at most n digits each */

> Generic hint for all algorithms problems:
>
> Imagine how you, personally, would do it,
> with a pencil and paper,
> when n = 100.

---

x = 12345678901234567890123456789012345678901234567890
y = 31415926535897932384626433832795028841971693993751

How would you add these two 50-digit numbers?

```
  000▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪ 01011111
  12345678901234567890123456789012345678901234567890
+ 31415926535897932384626433832795028841971693993751
  --------------------------------------------------
  437▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪ 2928561641
```

```
  000▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪ 01011111
  12345678901234567890123456789012345678901234567890
+ 31415926535897932384626433832795028841971693993751
  --------------------------------------------------
  437▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪ 2928561641
```

Add(x, y)
/* Assume x, y encoded as base-10 strings with array indices 0…$n-1$, least-significant-digit first, leading 0's included. We freely convert between digit characters and BoundedInts. */

carry ← 0
for i from 0 to $n-1$ do // i and *carry* are BoundedInts
  columnSum ← $x[i]$ + $y[i]$ + *carry* // also a BoundedInt
  $z[i]$ ← (*columnSum* mod 10)
  carry ← (*columnSum* − $z[i]$) ÷ 10
$z[n]$ ← *carry*
return z

Running time: Θ(n)

Add(x, y)
/* Assume x, y encoded as base-10 strings with array indices 0…$n-1$, least-significant-digit first, leading 0's included. We freely convert between digit characters and BoundedInts. */

carry ← 0
for i from 0 to $n-1$ do // i and *carry* are BoundedInts
  columnSum ← $x[i]$ + $y[i]$ + *carry* // also a BoundedInt
  $z[i]$ ← (*columnSum* mod 10)
  carry ← (*columnSum* − $z[i]$) ÷ 10
$z[n]$ ← *carry*
return z

Running time: Θ(n)

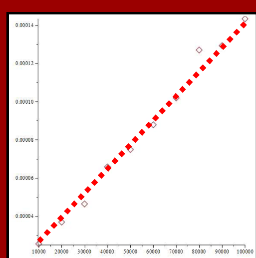
Could there be a fundamentally faster algorithm?

No. The output is ≥ n digits, so you must spend time ≥ n just to write down the answer.

So the *intrinsic complexity* of adding is Θ(n).

Time to do *x* + *y* in Python on your laptop,
when *x* = 22···222, *y* = 77···777,
n digits each, n = c·10^4, c = 1…10.

seconds



n

Exercise:
Write an algorithm for ***subtraction***:
doing *x*−*y* when *x* and *y* are n-digit BigInts.
Explain why your algorithm is Θ(n) time.

Onward to **multiplication**!

How long does it take to multiply integers?

Generic hint for all algorithms problems:

Imagine how you, personally, would do it,
with a pencil and paper,
when n = 100.

Generic hint for all problems:

Try small cases.

So let's actually try n = 4 first ☺

```
     2 1
      6 4 2 1
    × 5 2 1 3
    ---------
    1 9 2 6 3
    6 4 2 1
1 2 8 4 2
+ 3 2 1 0 5
-----------
  3 3 4 7 2 6 7 3
```

By the way, why does this work?

```
      6 4 2 1
    × 5 2 1 3
    ---------
    1 9 2 6 3
    6 4 2 1
1 2 8 4 2
+ 3 2 1 0 5
-----------
  3 3 4 7 2 6 7 3
```

$$(6\cdot10^3 + 4\cdot10^2 + 2\cdot10^1 + 1\cdot10^0) \times (5\cdot10^3 + 2\cdot10^2 + 1\cdot10^1 + 3\cdot10^0)$$
$$= (6\cdot3)\cdot10^{3+0} + (4\cdot3)\cdot10^{2+0} + (2\cdot3)\cdot10^{1+0} + (1\cdot3)\cdot10^{0+0}$$
$$+ (6\cdot1)\cdot10^{3+1} + (4\cdot1)\cdot10^{2+1} + (2\cdot1)\cdot10^{1+1} + (1\cdot1)\cdot10^{0+1}$$
$$+ (6\cdot2)\cdot10^{3+2} + (4\cdot2)\cdot10^{2+2} + (2\cdot2)\cdot10^{1+2} + (1\cdot2)\cdot10^{0+2}$$
$$+ (6\cdot5)\cdot10^{3+3} + (4\cdot5)\cdot10^{2+3} + (2\cdot5)\cdot10^{1+3} + (1\cdot5)\cdot10^{0+3}$$

$$
\begin{array}{r}
6\,4\,2\,1 \\
\times\; 5\,2\,1\,3 \\
\hline
{\scriptstyle 18\;\;12}\;6\;3 \\
6\;4\;2\;1\;\; \\
{\scriptstyle 12}\;8\;4\;2\;\;\;\; \\
+\quad {\scriptstyle 30\;\;20\;\;10}\;5\;\;\;\;\;\; \\
\hline
3\,3\,4\,7\,2\,6\,7\,3
\end{array}
$$

$(6\cdot 10^3 + 4\cdot 10^2 + 2\cdot 10^1 + 1\cdot 10^0) \times (5\cdot 10^3 + 2\cdot 10^2 + 1\cdot 10^1 + 3\cdot 10^0)$

$= (6\cdot 3)\cdot 10^{3+0} + (4\cdot 3)\cdot 10^{2+0} + (2\cdot 3)\cdot 10^{1+0} + (1\cdot 3)\cdot 10^{0+0}$

$+ (6\cdot 1)\cdot 10^{3+1} + (4\cdot 1)\cdot 10^{2+1} + (2\cdot 1)\cdot 10^{1+1} + (1\cdot 1)\cdot 10^{0+1}$

$+ (6\cdot 2)\cdot 10^{3+2} + (4\cdot 2)\cdot 10^{2+2} + (2\cdot 2)\cdot 10^{1+2} + (1\cdot 2)\cdot 10^{0+2}$

$+ (6\cdot 5)\cdot 10^{3+3} + (4\cdot 5)\cdot 10^{2+3} + (2\cdot 5)\cdot 10^{1+3} + (1\cdot 5)\cdot 10^{0+3}$

$$
\begin{array}{r}
6\,4\,2\,1 \\
\times\; 5\,2\,1\,3 \\
\hline
{\scriptstyle 18\;\;12}\;6\;3 \\
6\;4\;2\;1\;\; \\
{\scriptstyle 12}\;8\;4\;2\;\;\;\; \\
+\quad {\scriptstyle 30\;\;20\;\;10}\;5\;\;\;\;\;\; \\
\hline
3\,3\,4\,7\,2\,6\,7\,3
\end{array}
$$

Stage 1

for $i = 0 \ldots n-1$
  for $j = 0 \ldots n-1$
    $table[i][j] \leftarrow x[j]\cdot y[i]$

Stage 2

(add up the columns of *table*)

Stage 1 takes $\Theta(n^2)$ time.

Each *table*[*i*][*j*] is a BoundedInt (between 0…81)

Actually… How do you store a 2-d array??

(Technically, it's implemented by a 1-d array.
table[*i*][*j*] stored at *flatTable*[$n\cdot i + j$].
BoundedInt arithmetic to compute index;
note that it is between 0 and n^2-1.)

Stage 1

for $i = 0 \ldots n-1$
  for $j = 0 \ldots n-1$
    $table[i][j] \leftarrow x[j]\cdot y[i]$

Stage 2

(add up the columns of *table*)

Stage 2 takes $\Theta(n^2)$ time.

n columns, and $\Theta(n)$ time per column.

Carries are no longer just 1 digit;

can be as large as 9n. Column sums $\leq$ 90n.

Both storable as BoundedInt.

I leave the details of summing/carrying to you.

Stage 1

```
for i = 0…n−1
  for j = 0…n−1
    table[i][j] ← x[j]·y[i]
```

Stage 2

(add up the columns of *table*)

Running time of *this* multiplication algorithm: $\Theta(n^2)$

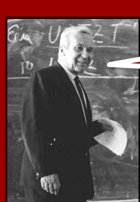
Could there be a fundamentally faster algorithm?

Seems like no… Yet, could we prove it?

The output is $\geq$ n digits, so you must spend $\Omega(n)$ time just to write down the answer.

There's still a gap.

Is the intrinsic complexity of integer multiplication quadratic or linear?

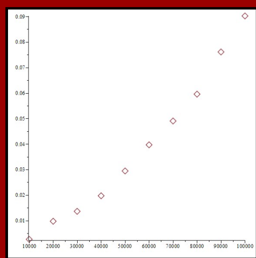


I conjecture it's quadratic, $\Omega(n^2)$.

Andrey Kolmogorov, 1960

Time to do $x * y$ in Python on your laptop,
when $x = 22\cdots222$, $y = 77\cdots777$,
n digits each, $n = c \cdot 10^4$, $c = 1 \ldots 10$.

seconds



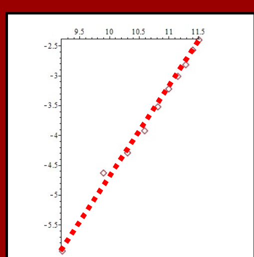
n

doesn't look linear…

is that a parabola…?

Time to do $x * y$ in Python on your laptop,
when $x = 22\cdots222$, $y = 77\cdots777$,
n digits each, $n = c \cdot 10^4$, $c = 1 \ldots 10$.

log(seconds)



log(n)

slope $\approx \dfrac{3.6}{2.3}$

≈ 1.57!?

Time to do $x * y$ in Python on a laptop,
when $x = 22\cdots222$, $y = 77\cdots777$,
n digits each, $n = c \cdot 10^4$, $c = 1 \ldots 10$.

Is Python doing an $\Theta(n^{1.57})$-time algorithm for multiplication?!

Yes!!

(Well, $\Theta(n^{1.58\ldots})$ actually.)

slope $\approx \dfrac{3.6}{2.3}$

≈ 1.57!?

Divide and conquer!

O_O

Anatoly Karatsuba, 1960

Karatsuba's Multiplication Algorithm

```
          6 4 2 1
        × 5 2 1 3
        ---------
        18 12 6 3
      6 4 2 1
   12 8 4 2
+ 30 20 10 5
-----------------
  3 3 4 7 2 6 7 3
```

$(6\cdot10^3 + 4\cdot10^2 + 2\cdot10^1 + 1\cdot10^0) \times (5\cdot10^3 + 2\cdot10^2 + 1\cdot10^1 + 3\cdot10^0)$

$= (6\cdot3)\cdot10^{3+0} + (4\cdot3)\cdot10^{2+0} + (2\cdot3)\cdot10^{1+0} + (1\cdot3)\cdot10^{0+0}$

$+ (6\cdot1)\cdot10^{3+1} + (4\cdot1)\cdot10^{2+1} + (2\cdot1)\cdot10^{1+1} + (1\cdot1)\cdot10^{0+1}$

$+ (6\cdot2)\cdot10^{3+2} + (4\cdot2)\cdot10^{2+2} + (2\cdot2)\cdot10^{1+2} + (1\cdot2)\cdot10^{0+2}$

$+ (6\cdot5)\cdot10^{3+3} + (4\cdot5)\cdot10^{2+3} + (2\cdot5)\cdot10^{1+3} + (1\cdot5)\cdot10^{0+3}$

Karatsuba's Multiplication Algorithm

$6421 \times 5213 = \cdots$

$(6\cdot10^3 + 4\cdot10^2 + 2\cdot10^1 + 1\cdot10^0) \times (5\cdot10^3 + 2\cdot10^2 + 1\cdot10^1 + 3\cdot10^0)$

Too much dividing, not enough conquering!

Let's try splitting both numbers into 2 halves.

Karatsuba's Multiplication Algorithm

$6421 \times 5213 = \cdots$

$(64 \cdot 10^2 + 21) \times (52 \cdot 10^2 + 13)$

$(64 \cdot 52) \cdot 10^4 + (64 \cdot 13 + 21 \cdot 52) \cdot 10^2 + (21 \cdot 13)$

Could compute $64 \cdot 52$ recursively!
(Multiplying two n/2-digit numbers.)

Could compute $21 \cdot 13$ recursively.

Could compute $64 \cdot 13$ & $64 \cdot 13$ recursively.

Karatsuba's Multiplication Algorithm

$6421 \times 5213 = \cdots$

Turns out: Splitting your numbers into 4 pieces,
then making 4 recursive calls,
still ends up giving quadratic time.

Could compute $64 \cdot 52$ recursively.
(Multiplying two n/2-digit numbers.)

Could compute $21 \cdot 13$ recursively.

Could compute $64 \cdot 13$ & $64 \cdot 13$ recursively.

Karatsuba's Multiplication Algorithm

$6421 \times 5213 = \cdots$

$(64 \cdot 52) \cdot 10^4 + (64 \cdot 13 + 21 \cdot 52) \cdot 10^2 + (21 \cdot 13)$

Can compute $64 \cdot 52$ and $21 \cdot 13$ recursively.

Karatsuba's brainwave:
Compute $(64-21) \cdot (52-13)$, recursively.

(Subtracting two n/2-digit numbers: $\Theta(n)$ time.
Doing *one* multiplication on n/2-digit numbers.)

Karatsuba's Multiplication Algorithm

$$6421 \times 5213 = \cdots$$

$$(64 \cdot 52) \cdot 10^4 + (64 \cdot 13 + 21 \cdot 52) \cdot 10^2 + (21 \cdot 13)$$

Can compute $64 \cdot 52$ and $21 \cdot 13$ recursively.

Karatsuba's brainwave:

Compute $(64 - 21) \cdot (52 - 13)$, recursively.

Gives you $64 \cdot 52 - 64 \cdot 13 - 21 \cdot 52 + 21 \cdot 13$.

Subtract off $64 \cdot 52$ & $21 \cdot 13$, negate, you get

$$64 \cdot 13 + 21 \cdot 52$$

Karatsuba's Multiplication Algorithm

$$6421 \times 5213 = \cdots$$

$$(64 \cdot 52) \cdot 10^4 + (64 \cdot 13 + 21 \cdot 52) \cdot 10^2 + (21 \cdot 13)$$

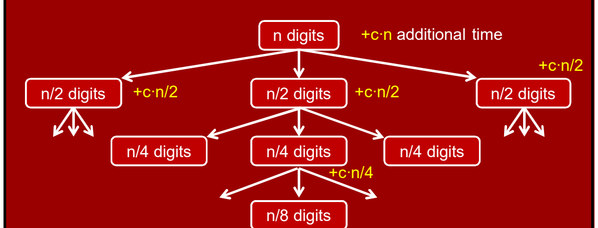
Compute $64 \cdot 52$ and $21 \cdot 13$ recursively.

Compute $(64 - 21) \cdot (52 - 13)$, recursively.

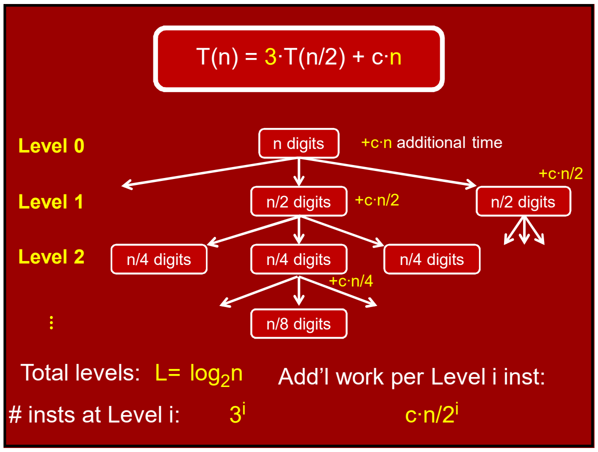
Now some additions and subtractions on n-digit numbers (time $\Theta(n)$) gives you answer.

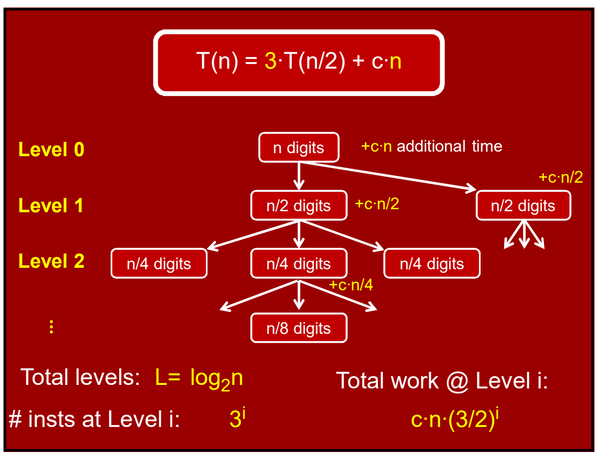
Recurrence: $\boxed{T(n) = 3 \cdot T(n/2) + c \cdot n}$

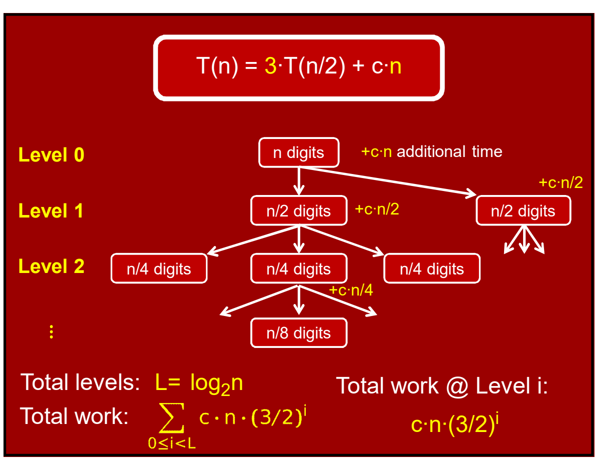
$$\boxed{T(n) = 3 \cdot T(n/2) + c \cdot n}$$



Total levels: $L = \log_2 n$


$T(n) = 3 \cdot T(n/2) + c \cdot n$
Level 0
Level 1
Level 2
⋮
n digits
+c·n additional time
n/2 digits
+c·n/2
n/2 digits
+c·n/2
n/4 digits
n/4 digits
n/4 digits
+c·n/4
n/8 digits
Total levels: $L = \log_2 n$
insts at Level i: 3^i
Add'l work per Level i inst:
$c \cdot n/2^i$


$T(n) = 3 \cdot T(n/2) + c \cdot n$
Level 0
Level 1
Level 2
⋮
n digits
+c·n additional time
n/2 digits
+c·n/2
n/2 digits
+c·n/2
n/4 digits
n/4 digits
n/4 digits
+c·n/4
n/8 digits
Total levels: $L = \log_2 n$
insts at Level i: 3^i
Total work @ Level i:
$c \cdot n \cdot (3/2)^i$


$T(n) = 3 \cdot T(n/2) + c \cdot n$
Level 0
Level 1
Level 2
⋮
n digits
+c·n additional time
n/2 digits
+c·n/2
n/2 digits
+c·n/2
n/4 digits
n/4 digits
n/4 digits
+c·n/4
n/8 digits
Total levels: $L = \log_2 n$
Total work: $\sum_{0 \le i < L} c \cdot n \cdot (3/2)^i$
Total work @ Level i:
$c \cdot n \cdot (3/2)^i$

$T(n) = 3 \cdot T(n/2) + c \cdot n$

Final running time:

$$\sum_{0 \le i < L} c \cdot n \cdot (3/2)^i = c \cdot n \cdot \frac{(3/2)^L - 1}{(3/2) - 1} \le c \cdot n \cdot \frac{(3/2)^{\log_2 n}}{1/2}$$

$$= 2c \cdot n \cdot n^{\log_2 (3/2)} = 2c \cdot n^{\log_2 3}$$

$$= \Theta(n^{\log_2 3}) \qquad !!$$

$\log_2 3 \approx 1.58\ldots$

Total levels: $L = \log_2 n$

Total work: $\sum_{0 \le i < L} c \cdot n \cdot (3/2)^i$

How long does it take to multiply integers?

Grade school algorithm: $\Theta(n^2)$

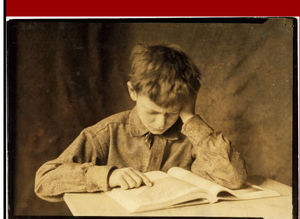
Karatsuba's algorithm: $\Theta(n^{\log_2 3}) \approx \Theta(n^{1.58\ldots})$

Python actually uses this!

Can we do better?

Stay tuned for Lecture 25…!

Study Guide



Understand:

- RAM model
- Difference between BoundedInts & BigInts

Basic arithmetic:

- Why addition is $\Theta(n)$
- Why subtraction is $\Theta(n)$
- Why multiplication is $O(n^2)$
- Karatsuba: why multiplication is also $O(n^{\log_2 3})$