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

Computational Arithmetic





- 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





C-like pseudocode $lo \leftarrow 1; hi \leftarrow n$ while (lo < hi) if $x[lo] \neq x[hi]$ then REJECT $lo \leftarrow lo + 1; hi \leftarrow hi - 1$ ACCEPT

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

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

reverse of string *x*

Python



Turing Machine



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

C-like pseudocode $lo \leftarrow 1; hi \leftarrow n$ while (lo < hi)if $x[lo] \neq x[hi]$ then REJECT $lo \leftarrow lo + 1; hi \leftarrow hi - 1$ ACCEPT

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

Python

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



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

C-like pseudocode

Turing

Machine

 $lo \leftarrow 1; hi \leftarrow n$ while (lo < hi) if $x[lo] \neq x[hi]$ then REJECT $lo \leftarrow lo + 1; hi \leftarrow hi - 1$ ACCEPT

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

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

Python

T(n

Turing Machine



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

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

Not very realistic.

Python

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

Python

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

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

?

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

Maybe just solve any language L with import solveStuff solveStuff.solveL(x)

Python

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

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

Not even a question of fairness...

Time to do $x_{rev} = x[::-1]$ on your laptop, n = c·10⁶, c = 1...10.





Similarly for doing x == x_rev

seconds

n

Why does...

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

...seem to take $\Theta(n)$ time?

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

> for *i* from 1 to *n* $x_rev[i] \leftarrow x[n-i]$ for *i* from 1 to *n* if $x[i] \neq x_rev[i]$ then REJECT ACCEPT

- "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 ③

IsPalindrome(x) $lo \leftarrow 1$ $hi \leftarrow n$ 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.



You may assume the code knows input length, n.

IsPalindrome(x) // $x \in \Sigma^*$ is an array of characters $lo \leftarrow 1$ $hi \leftarrow n$ // n is the length of xloop while (lo < hi) if $x[lo] \neq x[hi]$ hen 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.



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

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

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



Arithmetica

Pythagorus

Allegory of Arithmetic by Gregor Reisch, 1503.

"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."

"Million-digit numbers?! Those variables *Io* 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."

"Million-digit numbers?! Those variables *Io* 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!"

"Million-digit numbers?! Those variables *Io* 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."

"C'mon. \heartsuit_{\bigcirc} . Be reasonable. $i \leftarrow 1$ while $i \leq n$ $i \leftarrow i+1$ has to be $\Theta(n)$ 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 n² or n³) 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).

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".

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

Io 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



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 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 = 12345678901234567890123456789012345678901234567890y = 31415926535897932384626433832795028841971693993751

How would you add these two 50-digit numbers?

12345678901234567890123456789012345678901234567890

+ 31415926535897932384626433832795028841971693993751

12345678901234567890123456789012345678901234567890 + 31415926535897932384626433832795028841971693993751

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. */

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

Running time: $\Theta(n)$

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. */

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

Running time: $\Theta(n)$

Could there be a fundamentally faster algorithm?

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

So the *intrinsic complexity* of adding is $\Theta(n)$.

Time to do x + y in Python on your laptop, when $x = 22 \cdots 222$, $y = 77 \cdots 777$, <u>n digits each</u>, $n = c \cdot 10^4$, c = 1...10.



n

Exercise:

Write an algorithm for *subtraction*: doing x-y when x and y are n-digit BigInts. Explain why your algorithm is $\Theta(n)$ 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 \bigcirc



By the way, why does this work?

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

33472673

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

33472673

				6	54	2	1
		_	×	5	52	1	3
				18	12	6	3
			6	4	2	1	
		12	8	4	2		
+	30	20	10	5			

	6421
	× 5213
	18 12 6 3
	6421
	¹² 8 4 2
-	+ 30 20 10 5
_	33472673
Stage 1	Stage 2
for $i = 0n-1$ for $j = 0n-1$	(add up the columns of <i>table</i>

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...n-1for j = 0...n-1table[i][j] $\leftarrow x[j] \cdot y[i]$ Stage 2 (add up the columns of *table*) Stage 2 takes Θ(n²) time.
n columns, and Θ(n) time per column.
Carries are no longer just 1 digit;
can be as large as 9n. Column sums ≤ 90n.
Both storable as BoundedInt.
I leave the details of summing/carrying to you.

Stage 1 for i = 0...n-1for j = 0...n-1 $table[i][j] \leftarrow x[j] \cdot y[i]$ Stage 2 (add up the columns of *table*) Running time of thismultiplication 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?



Andrey Kolmogorov, 1960

Time to do x * y in Python on your laptop, when $x = 22 \cdots 222$, $y = 77 \cdots 777$, n digits each, $n = c \cdot 10^4$, c = 1...10.



Time to do x * y in Python on your laptop, when $x = 22 \cdots 222$, $y = 77 \cdots 777$, n digits each, $n = c \cdot 10^4$, c = 1...10.



Time to do x * y in Python on a laptop, when $x = 22 \cdots 222$, $y = 77 \cdots 777$, n digits each, $n = c \cdot 10^4$, c = 1...10.

Is Python doing an $\Theta(n^{1.57})$ -time algorithm for multiplication?! $slope \approx \frac{3.6}{2.3}$ Yes!! $\approx 1.57!$? (Well, $\Theta(n^{1.58...})$ actually.)

Divide and conquer!





Anatoly Karatsuba, 1960

Karatsuba's Multiplication Algorithm 6421 $\times 5213$ 18 12 6 3 642 1 12 8 4 2+ 30 20 10 5

33472673

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

6421 × 5213 = ···

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

Too much dividing, not enough conquering!

Let's try splitting both numbers into 2 halves.

6421 × 5213 = …

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

 $(64.52) \cdot 10^4 + (64.13 + 21.52) \cdot 10^2 + (21.13)$

Could compute 64·52 recursively! (Multiplying two n/2-digit numbers.) Could compute 21·13 recursively. Could compute 64·13 & 64·13 recursively.

6421 × 5213 = …

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

Could compute 64.52 recursively. (Multiplying two n/2-digit numbers.)

Could compute 21.13 recursively.

Could compute 64.13 & 64.13 recursively.

Karatsuba's Multiplication Algorithm 6421 × 5213 = ··· $(64.52) \cdot 10^4 + (64.13 + 21.52) \cdot 10^2 + (21.13)$ Can compute 64.52 and 21.13 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.)

6421 × 5213 = ···

 $(64.52) \cdot 10^4 + (64.13 + 21.52) \cdot 10^2 + (21.13)$

Can compute 64.52 and 21.13 recursively.

Karatsuba's brainwave: Compute $(64-21) \cdot (52-13)$, recursively. Gives you 64.52 - 64.13 - 21.52 + 21.13. Subtract off 64.52 & 21.13, negate, you get 64.13 + 21.52

Karatsuba's Multiplication Algorithm 6421 × 5213 = ··· $(64 \cdot 52) \cdot 10^4 + (64 \cdot 13 + 21 \cdot 52) \cdot 10^2 + (21 \cdot 13)$ Compute 64.52 and 21.13 recursively. Compute $(64-21) \cdot (52-13)$, recursively. Now some additions and subtractions on n-digit numbers (time $\Theta(n)$) gives you answer.

Recurrence:

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



Total levels: L= log₂n







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

Final running time:

log₂3 ≈ 1.58....

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: Θ(n^{log₂3}) ≈ Θ(n^{1.58...}) 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 $O(n^{\log_2 3})$