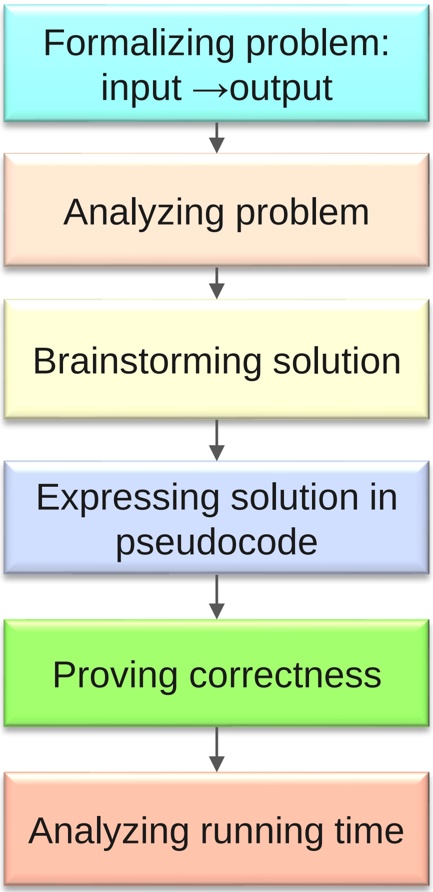
Algorithm design: basic tools

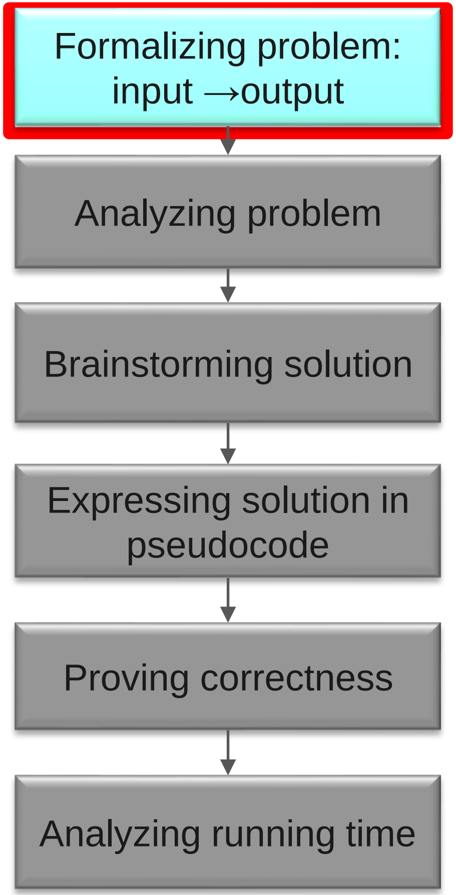
Lecture 01.02

by Marina Barsky

Algorithm design flowchart



Algorithm design flowchart



Formalizing problem

- Problem vs. problem instance:

  Take: 24 and 15. What is their greatest common divisor (gcd)?

- Formalized general problem: input and output

  Input: 2 integer numbers *a* and *b*

  Output: The greatest common divisor *gcd(a,b)*

Problem: Compute GCD

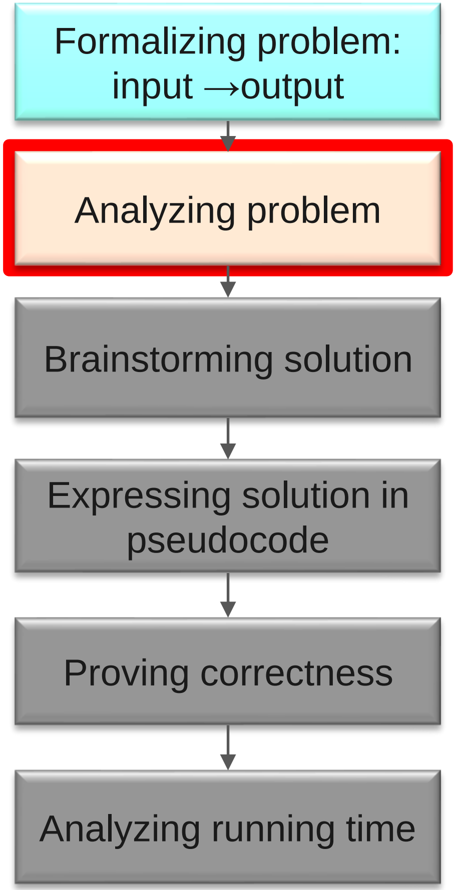
Input: 2 integers $a,\ b > 0,\ a > b$

Output: $\mathrm{gcd}(a,\ b)$.

We want it to work on large numbers:

$\mathrm{gcd}(3918848,\ 1653264)$ ← **Problem instance**

Algorithm design flowchart



Analyzing: Greatest Common Divisor

Formal Definition

For integers, *a* and *b*, their ***greatest common divisor*** or ***gcd(a, b)*** is the largest integer *d* s.t. *d* divides both *a* and *b (without remainder)*.

Why do we want to compute it:

- Put fraction a/b into simplest form.
- Need to check remainders of (a/d) (b/d)
  - *d* should divide both *a* and *b*.
  - Want *d* to be as large as possible.

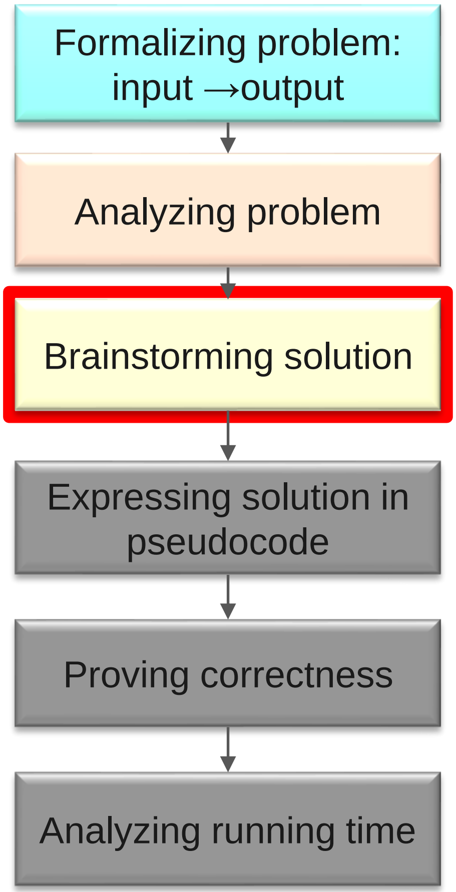
a=15, b=45

both 15 and 45 are divisible by 3, 5,15

we want to find 15

Go over an example

Algorithm design flowchart



Brainstorming

Problem: Compute GCD

Input: 2 integers $a,\ b > 0$, $a > b$

Output: $\gcd(a,\ b)$.

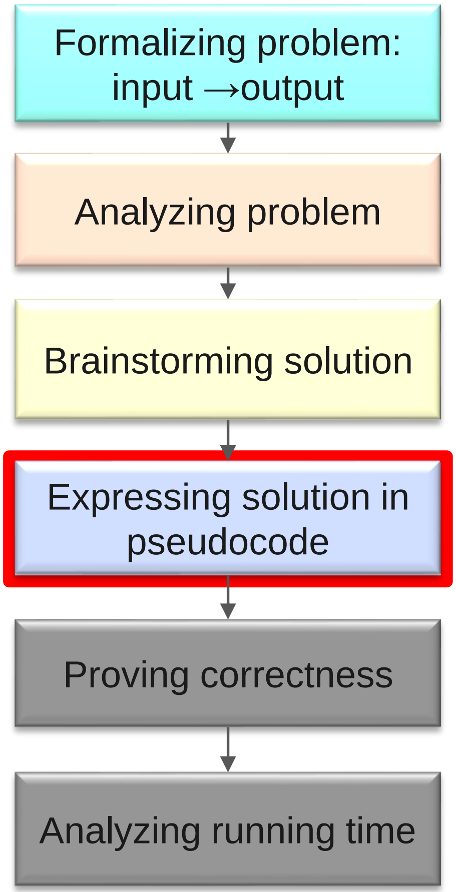
According to the problem and the definition of gcd:

- We need to go over integers 1, 2, ...
- Check if each such integer *i* divides both *a* and *b* without remainder
- Keep the largest such number
- Stop when $i = min(a,b) = b$

a=15, b=45

both 15 and 45 are divisible by 3, 5,15

Algorithm design flowchart



Three ways of describing algorithmic solutions

- English
- Pseudocode
- Program

Increasing precision

Pseudocode: example

```
FOR i ← 1 TO 100 DO
    IF i is divisible by 3 AND i is divisible by 5 THEN
        OUTPUT "Both"
    ELSE IF i is divisible by 3 THEN
        OUTPUT "By 3"
    ELSE IF i is divisible by 5 THEN
        OUTPUT "By 5"
    ELSE
        OUTPUT i
```

Pseudocode: example

```
FOR i ← 1 TO 100 DO
    IF i is divisible by 3 AND i is divisible by 5 THEN
        OUTPUT "Both"
    ELSE IF i is divisible by 3 THEN
        OUTPUT "By 3"
    ELSE IF i is divisible by 5 THEN
        OUTPUT "By 5"
    ELSE
        OUTPUT i
```

Python equivalent

```python
def some_algorithm ():
    for i in range(1,101):
        if i%3 == 0 and i%5 == 0:
            print(i, "Both")
        elif i%3 == 0:
            print(i, "By 3")
        elif i%5 == 0:
            print(i, "By 5")
        else:
            print(i)
```

Pseudocode does not have specific syntax: it just has to be clear and unambiguous

Some specifics

- Assignment operator:
  X := 5
  X ← 5
- Comparing for equality:
  if x = y
- *FOR* loop:
  for each element x in sequence:
  for i from 1 to n:
  for i from 1 to n step 2:
  for i from n down to 1:
- *WHILE* loop:
  same as if

Pseudocode does not have specific syntax

But keep in mind <u>the goal</u>:
pseudocode **must be easily translatable into a working program** (in any language).

Pseudocode for GCD

English:

Try every integer from 1 to min(*a, b*).
If the integer divides both *a* and *b*, remember the best *gcd* so far.
Since the integers we test are increasing,
the algorithm will remember the best – the greatest common divisor for *a* and *b.*

Pseudocode:

Algorithm NaiveGCD(*a, b*)

```
best ← 1
for d from 2 to min(a, b):
    if d|a and d|b:
        best ← d
return best
```

Pseudocode and code for GCD

Pseudocode:

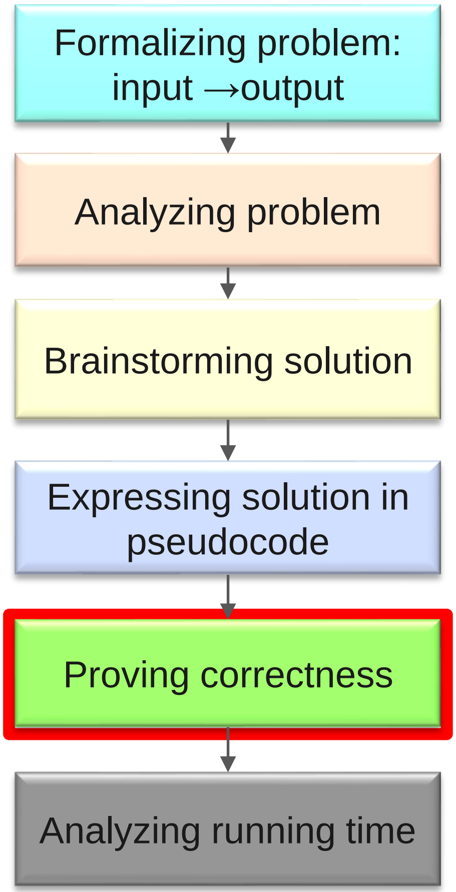
Algorithm NaiveGCD(a, b)

$best \leftarrow 1$
for d from 2 to min(a, b):
  if $d | a$ and $d | b$:
    $best \leftarrow d$
return $best$

Code:

```
def gcd_naive(a, b):
    best = 1
    for i in range (2, min(a,b)+1):
        if a%i == 0 and b%i == 0:
            best = i
    return best
```

Algorithm design flowchart



Correctness proof: iterative algorithms

For iterative algorithms (algorithms which use loops) -
we need to check:

- ***Initialization***: states what must be true before entering a loop
- ***Loop invariant***: states what property must be preserved with each iteration of the loop
- ***Termination***: states what must be true after exiting the loop

Example

- Design an algorithm that takes as an input number n, and computes the sum of all positive integers *i* such that $i^3 < n$

```
Algorithm one(n):
    sum ← 0
    i ← 1;

    while (i * i * i < n) do:
        sum ← sum + i
        i ← i + 1

    return sum
```

```
Algorithm two(n):
    sum ← 0
    i ← 1;

    do:
        sum ← sum + i
        i ← i + 1
    while (i * i * i < n)

    return sum
```

Proving correctness of Algorithm *one*

```
Algorithm one(n):
    sum ← 0
    i ← 1;

    while (i * i * i < n) do:
        sum ← sum + i
        i ← i + 1

    return sum
```

- ***Initialization***: *sum* is zero, *i* is set to the first positive integer. ✓
- ***Loop invariant***: After each iteration *i sum* must contain the sum of all numbers from 1 to *i* subject to constraint: $i^3 < n$. ✓
- ***Termination***: we exit the loop when $i^3 >= n$. At this point *sum* contains the sum of the required numbers. If n=1, we did not enter the loop, and the *sum* correctly remains zero. ✓

CORRECT

Proving correctness of Algorithm *two*

```
Algorithm two(n):
    sum ← 0
    i ← 1;

    do:
        sum ← sum + i
        i ← i + 1
    while (i * i * i < n)

    return sum
```

- ***Initialization***: *sum* is zero, *i* is the first positive integer.
- ***Loop invariant***: we enter the loop **without checking loop condition** and add *i* to *sum*. If *n*=1, we still enter the loop, and the *sum* is 1, but is should be 0! We check if $i^3 < n$ **after the addition is already performed**. After each iteration of the loop, *sum* contains the sum of all numbers from 1 to *i*, even if *i* already violated the condition! Loop invariant is violated.

INCORRECT!

Correctness proof: iterative algorithms

For iterative algorithms (algorithms which use loops)
we need to check:

- ***Initialization***: states what must be true before entering a loop
- ***Loop invariant***: states what property must be preserved with each iteration of the loop
- ***Termination***: states what must be true after exiting the loop

- If we know that the algorithm is designed strictly according to the problem definition, we can skip the proof step

Correctness proof: recursive algorithms

To prove the correctness of recursive algorithms we use proof by mathematical induction!

- ***Base case***: check if the stopping condition correctly computes the base case.
- ***Assumption:*** the algorithm is correct for $n = k - 1$.
- ***Given the base and the assumption:*** prove that it is correct for $n = k$.

Example

- Design an algorithm which takes as an input string *s* of length *n* and returns a new string where the characters of *s* appear in a reverse order.

```
Algorithm reverse(s of length n)
    if n = 0 then
        return ε
    else
        a ← s[n]
        r ← s - a
        return a + reverse(r)
```

Correctness proof: *reverse*

```
Algorithm reverse(s of length n)
    if n = 0 then
        return ε
    else
        a ← s[n]
        r ← s - a
        return a + reverse(r)
```

It seems natural to do induction on n, the length of the string.

Base case: if $n = 0$, $s = \varepsilon$, the empty string. In this case, the first return statement is executed, and the algorithm returns ε, the correct reversal of itself.
In other words, *reverse*(ε) = ε.

Hypothesis: Suppose as inductive hypothesis that, for any string of length $k - 1$, *reverse*($c_1c_2c_3 \ldots c_{k-1}$) = $c_{k-1}c_{k-2} \ldots c_2c_1$, for some $k > 0$.

Proof: Now suppose *reverse* is sent a string of length k. Then:
reverse($c_1c_2c_3 \ldots c_{k-1}\, c_k$)
= c_k *reverse*($c_1c_2c_3 \ldots c_{k-1}$) (according to the algorithm)
= $c_k\, c_{k-1}c_{k-2} \ldots c_2c_1$(by inductive hypothesis)

■

Correctness of *NaiveGCD*

```
Algorithm NaiveGCD(a, b)

best ← 1
for d from 2 to min(a, b):
    if d|a and d|b:
        best ← d
return best
```

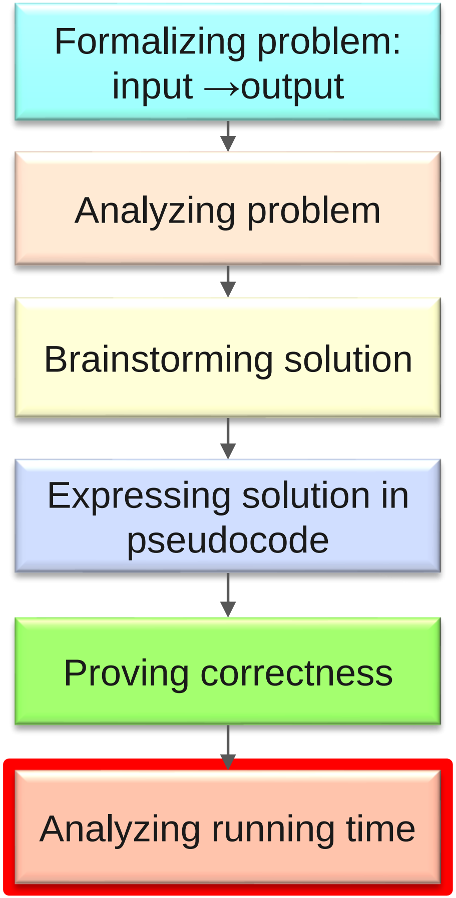
We assume without loss of generality that $a>b$.

- ***Initialization***: *best* is set to 1. This is correct because every integer is divisible by 1.
- ***Loop invariant***: we check if the next *d* can be a common divisor, and if yes, update *best*. Thus after each iteration *best* contains the greatest common divisor from 1 to *d*.
- ***Termination***: according to loop invariant, *best* contains the greatest among all common divisors from 1 to min(*a, b*).

  Should we check numbers > min(*a, b*)?

  No, because if b < a, say, we divide *b* by $d>b$, and the remainder always will be *b* (non-zero) ■

Algorithm design flowchart



How long does it take to compute?

How many steps does your algorithm take?

The pseudocode makes it easy to count the total number of steps as it relates to the input size *n* and the nature of the input

Algorithm has_divisors(n)

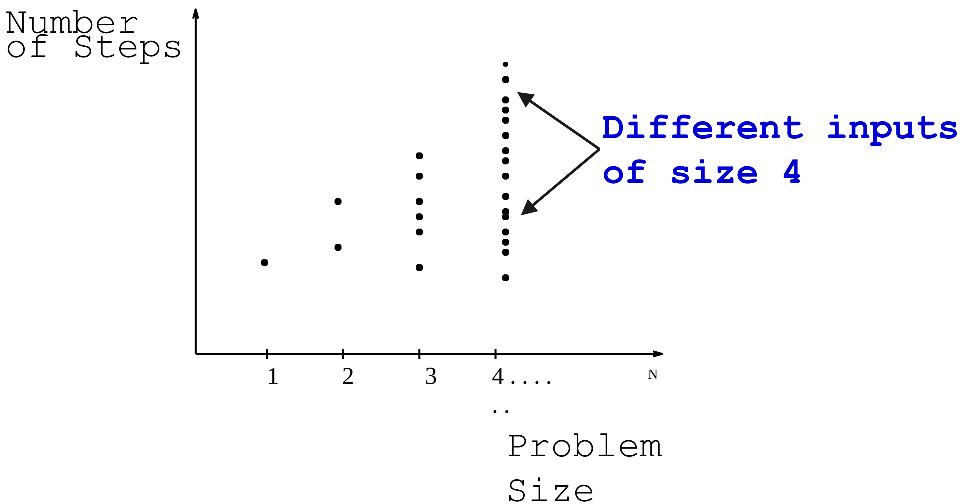
```
for i from 2 to n-1:
    if i|n :
        return True
return False
```

- It may happen that algorithm produces True already on the first operation, because *n* is even: 1 operation in total

- However, it may take *n* - 2 steps in case that *n* is prime: *n* - 2 operations in total

Number of operations vs. input size

- We can count number of steps for a variety of inputs and for different values of n and plot the results



RAM model of computation

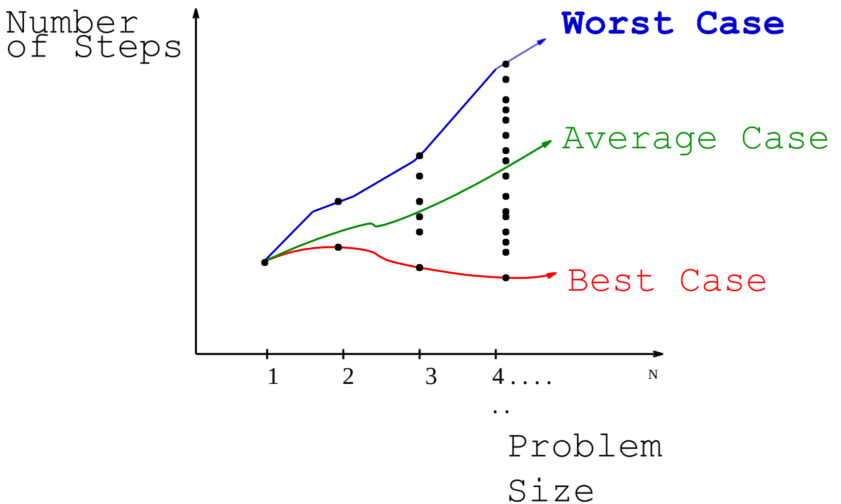
This process of counting computer operations is greatly simplified if we accept **the RAM model of computation**:

- Access to each memory element takes a constant time (1 step)
- Each "simple" operation (+, -, =, if, call) takes 1 step.
- Loops and subroutine calls are *not* simple operations: they depend upon the size of the data and the contents of a subroutine:
  - "sort()" is not a single-step operation
  - "max(list)" is not a single-step operation
  - " if x in list" is not a single-step operation

This model is useful and accurate in the same sense as the **flat-earth model** (which ***is*** useful)!

Number of operations vs. input size

- We want to be able to describe the performance of our algorithm in more general terms
- We see that there is the best case and the worst case for each value of *n*



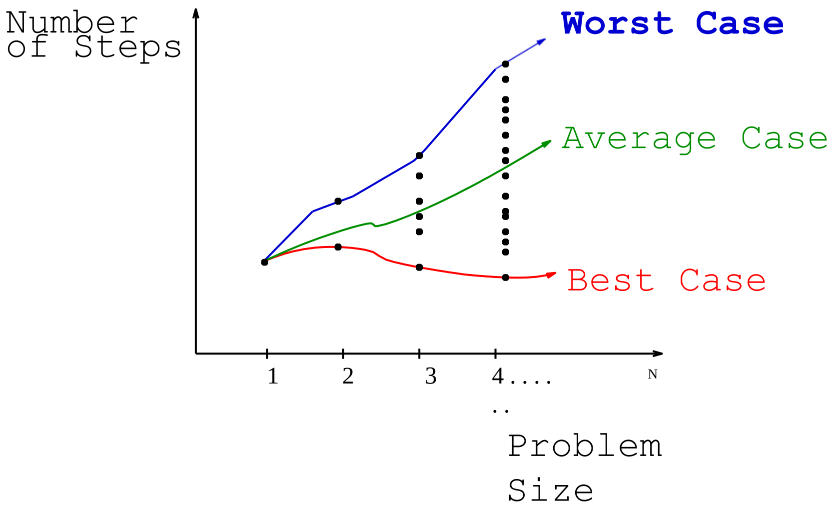
Complexity

- The best case complexity of an algorithm is the function defined by the minimum number of steps taken on any instance of size *n*.
- The average-case complexity of the algorithm is the function defined by an average number of steps taken on any instance of size *n*.
- The worst case complexity of an algorithm is the function defined by the maximum number of steps taken on any instance of size *n*.

- Each of these complexities defines a **numerical function**: number of operations vs. size of the input

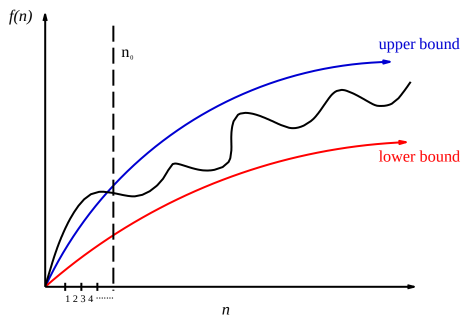
We are more interested in the worst case

Because the nature of the input is generally not known in advance, we concentrate on the worst-case: we want to know if it is practical to run this algorithm on large inputs of unknown nature



Still exact analysis is hard!

Best, worst, and average case are all difficult to deal with because the ***precise*** function details may be complicated:



It is easier to talk about *upper* and *lower bounds* of a function.

Asymptotic notation (*O*, Θ, Ω) allows us to describe complexity functions in practice.

Bounding Functions

- $f(n)$ = ***O***$(g(n))$ means $C \times g(n)$ is an ***upper bound*** on $f(n)$
- $f(n) = \mathbf{\Omega}(g(n))$ means $C \times g(n)$ is a ***lower bound*** on $f(n)$
- $f(n) = \mathbf{\Theta}(g(n))$ means $C_1 \times g(n)$ is a ***lower bound*** on $f(n)$ and $C_2 \times g(n)$ is an ***upper bound*** on $f(n)$

C, C_1, and C_2 are all constants independent of n.

Formal Definitions

Big-Oh

- $f(n) = \mathbf{O}(g(n))$ if there are positive constants n_0 and c such that to the right of n_0, the value of $f(n)$ always lies on or below $c \cdot g(n)$.

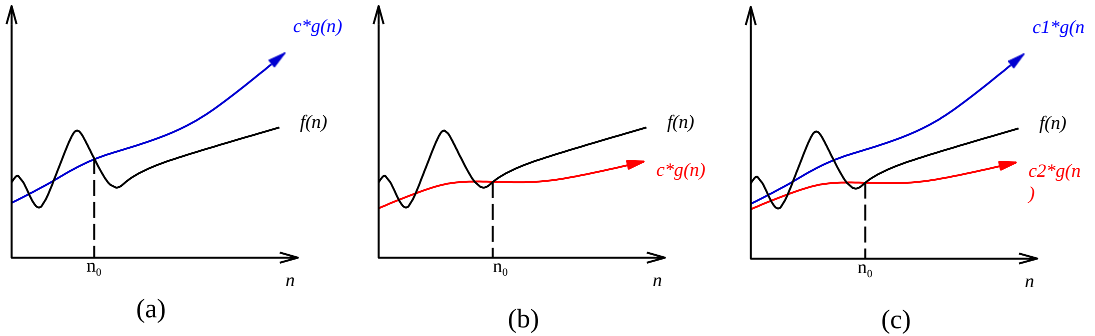
Omega

- $f(n) = \mathbf{\Omega}(g(n))$ if there are positive constants n_0 and c such that to the right of n_0, the value of $f(n)$ always lies on or above $c \cdot g(n)$.

Theta

- $f(n) = \mathbf{\Theta}(g(n))$ if there exist positive constants n_0, c_1, and c_2 such that to the right of n_0, the value of $f(n)$ always lies between $c_1 \cdot g(n)$ and $c_2 \cdot g(n)$ inclusive.

O, Ω, and Θ



- The definitions imply a constant n_0 beyond which they are satisfied.
- We do not care about small values of *n*.

Complexity of *NaiveGCD*

Algorithm NaiveGCD(a, b)

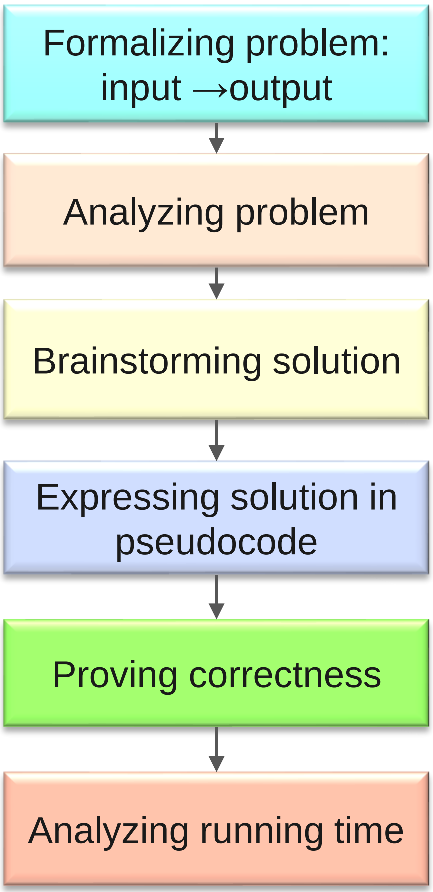
$best \leftarrow 0$	1 step
`for` d `from` 1 `to min(`a, b`):`	b steps
`if` $d \mid a$ `and` $d \mid b$:	1 step
$best \leftarrow d$	1 step
`return` $best$	1 step

- If a > b:

  Total steps: 1 + 2b + 1
- As *b* becomes bigger, we can ignore constants
- The NaiveGCD algorithm runs in time O(*b*)

We want it to work on large numbers:
gcd(3918848, 1653264)

Algorithm design flowchart



Are we done?

Algorithm designer mantra

"Perhaps the most important principle for the good algorithm designer is to **refuse to be content**"
Aho, Hopcroft, Ullman: "The design and Analysis of Computer Algorithms", 1974

- Structure of the input
- New insight
- Idea

Mantra: **Can we do better?**

Runtime of *NaiveGCD*

Pseudocode:

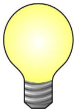
```
Algorithm NaiveGCD(a, b)

best ← 0
for d from 1 to min(a, b):
    if d|a and d|b:
        best ← d
return best
```

The NaiveGCD algorithm runs in time O(*b*)

We want it to work on large numbers:
gcd(3918848, 1653264)

Can we do better?

Euclid's observation



Theorem

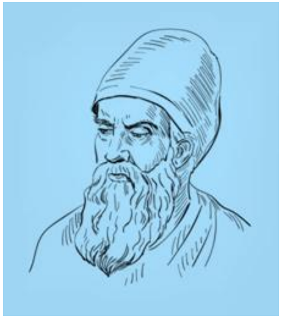
Let a > b, and *rem* be the remainder when *a* is divided by *b.*

Then

gcd(a, b) = gcd(rem, b) = gcd(b, rem).

Proof (sketch)

- $a = bq + rem$ for some integer q
- d divides a and b if and only if it divides rem and b



Euclid
Mid-4th - Mid-3rd
century BC

Euclidean GCD algorithm

Algorithm EuclidGCD(a, b)

if $b = 0$: return a

$rem \leftarrow$ the remainder when a is divided by b

return EuclidGCD(b, rem)

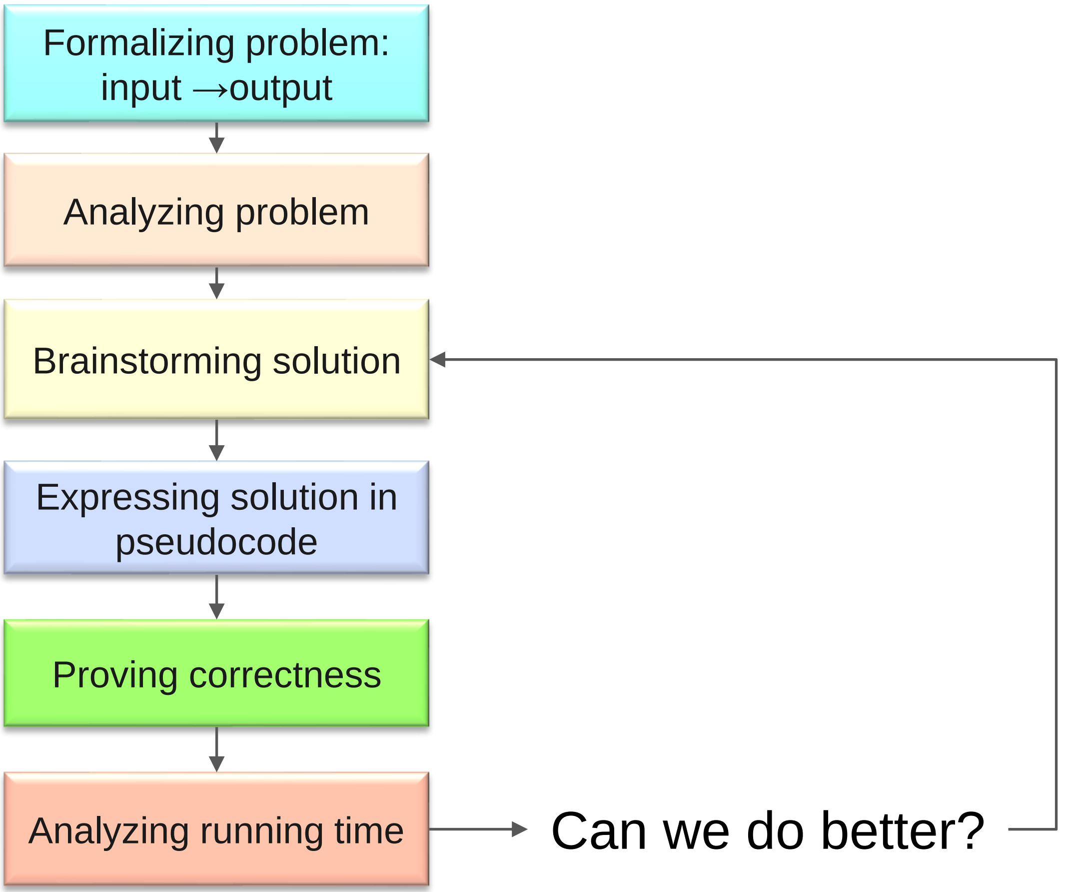
Example

gcd(33, 27)	33 % 27 = 6
gcd(27, 6)	27 % 6 = 3
gcd(6, 3)	6 % 3 = 0
gcd(**3**, 0)	

gcd(33, 27) = 3

- Each step reduces the size of numbers by about a factor of 2.
- Takes about $\log(ab)$ steps.
- GCDs of 100-digit numbers takes about 600 steps.
- Each step - a single integer division.

Algorithm design: infinite loop



Algorithms for Generating primes

Class activity

Sample problem: checking number for primality

Problem: divisors

Input: integer *n* > 0
Output: *True* if *n* is divisible by any number other than 1 and *n*, *False* otherwise

Algorithm has_divisors(n)

Write a naive algorithm which tests if *n* is divisible by any number between 2 and *n*-1.

Complexity of *has_divisors*

Upper bound ?

Lower bound ?

Can we do better?

has_divisors revisited

Problem: divisors

Input: integer n > 0
Output: True if n is divisible by any number other than 1 and n, False otherwise

Algorithm has_divisors2(n)

16 = 2*8

16 = 4*4

16 = 8*2

Do we really need to check all the n - 2 values?

Sample problem: generating primes

Problem: list of primes

Input: integer $n > 0$
Output: list of all prime numbers <= n

Algorithm list_primes(n)

Design a naive algorithm which generates a list of primes
Use algorithm *has_divisors2*

Sample problem: generating primes

Problem: list of primes

Input: integer $n > 0$
Output: list of all prime numbers <= n

Algorithm list_primes(n)

Design a naive algorithm which generates a list of primes

Use algorithm *has_divisors2*

What is worst-case complexity of this algorithm?

Sieve of Eratosthenes

Problem: list of primes

Input: integer $n > 0$
Output: list of all prime numbers <= n

	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20
21	22	23	24	25

Eratosthenes
276 – 194 BC

Sieve of Eratosthenes

Problem: list of primes

Input: integer *n* > 0
Output: list of all prime numbers <= *n*

	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20
21	22	23	24	25

2 is a prime. What do we know about 4, 6, 8…?

Sieve of Eratosthenes

Problem: list of primes

Input: integer $n > 0$
Output: list of all prime numbers <= n

	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20
21	22	23	24	25

4, 6, 8 are removed from the list

Sieve of Eratosthenes

Problem: list of primes

Input: integer $n > 0$
Output: list of all prime numbers <= n

	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20
21	22	23	24	25

Next prime is 3. What do we know about 9, 15, 21?

Sieve of Eratosthenes

Problem: list of primes

Input: integer *n* > 0
Output: list of all prime numbers <= *n*

	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20
21	22	23	24	25

We exclude 9, 15, 21

Sieve of Eratosthenes

Problem: list of primes

Input: integer $n > 0$
Output: list of all prime numbers <= n

	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20
21	22	23	24	25

We exclude remaining multiples of 5 - which is 25

Sieve of Eratosthenes

Problem: list of primes

Input: integer $n > 0$
Output: list of all prime numbers <= n

	2	3		5
	7		9	
11		13	14	15
	17		19	
21	22	23		25

Remaining numbers are all primes

Sieve of Eratosthenes: ancient version

Problem: list of primes

Input: integer $n > 0$
Output: list of all prime numbers <= n

Algorithm sieve_primes(n)

```
candidates←list [2,3….n]
primes←empty list
while candidates is not empty:
    p ← first element of candidates
    primes←primes + p
    candidates←candidates - p
    for each remaining number x in candidates:
        if x is divisible by p:
            candidates←candidates - x
return primes
```

Sieve of Eratosthenes: ancient version

Problem: list of primes

Input: integer $n > 0$
Output: list of all prime numbers <= n

Algorithm sieve_primes(n)

```
candidates←list [2,3….n]
primes←empty list
while candidates is not empty:
    p ← first element of candidates
    primes←primes + p
    candidates←candidates - p
    for each remaining number x in candidates:
        if x is divisible by p:
            candidates←candidates - x
return primes
```

Problem!

The removal of elements from the array: to remove an element from the middle of the array we need to move O(n) elements to fill the gap after removal

Sieve of Eratosthenes: ancient version

Problem: list of primes

Input: integer $n > 0$
Output: list of all prime numbers <= n

Algorithm sieve_primes(n)

candidates←list [2,3….n]
primes←empty list
while candidates is not empty:
    p ← first element of candidates
    primes←primes + p
    candidates←**candidates - p**
    for each remaining number x in candidates:
        if x is divisible by p:
            candidates←**candidates - x**
return primes

With the linked list: running time is $O(n^2)$

Can we do better?

Possible solution

Store the numbers in the doubly-linked list.
The removal will take time O(1).
But there will be a significant memory overhead.

Sieve of Eratosthenes: better implementation

Algorithm sieve_primes1(n)

```
candidates←list [1,2,3….n]
primes←empty list
for i from 2 to n:
    if candidates[i] > 0
        p = candidates[i]
        primes←primes + p
        for j=i+p to n step p:
            candidates[j] = 0
return primes
```

Not removing anything, just marking
We do not need to scan all the remaining elements – we can jump to elements of interest directly.

Sieve of Eratosthenes: complexity

Algorithm sieve_primes1(n)

```
candidates←list [1,2,3….n]
primes←empty list
for i from 2 to n:                    ← Outer loop
    if candidates[i] > 0
        p = candidates[i]
        primes←primes + p
        for j=i+p to n step p:        ← Inner loop
            candidates[j] = 0
return primes
```

p=2, the inner loop will be executed → (n/2) times

p=3, the inner loop will be executed → (n/3) times

p=5, the inner loop will be executed → (n/5) times

…

p=n, the inner loop will be executed → 1 time

Sieve of Eratosthenes: upper bound

Algorithm sieve_primes1(n)

```
candidates←list [1,2,3….n]
primes←empty list
for i from 2 to n:
    if candidates[i] > 0
        p = candidates[i]
        primes←primes + p
        for j=i+p to n step p:
            candidates[j] = 0
return primes
```

Outer loop

Inner loop

This is called amortized running time

So the total number of steps in the inner loop (**over the entire algorithm**):

$n/2 + n/3 + n/5 + \ldots + n/n$

Factor n out and you get:

$n(1/2 + 1/3 + 1/5 + \ldots + 1/n)$

What is the upper bound of this sum?

Sieve of Eratosthenes: upper bound

Algorithm sieve_primes1(n)

```
candidates←list [1,2,3….n]
primes←empty list
for i from 2 to n:
    if candidates[i] > 0
        p = candidates[i]
        primes←primes + p
        for j=i+p to n step p:
            candidates[j] = 0
return primes
```

The total number of steps:
n(1/2 + 1/3 + 1/5 +… + 1/n)

$$\sum_{n=1}^{\infty} \frac{1}{n} = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \cdots.$$

1/2 + 1/3 + 1/5 +… + 1/n < **1/2 + 1/3 + 1/4 + 1/5 +… + 1/n**

This is the sum of harmonic series: LINK

Sieve of Eratosthenes: upper bound

Algorithm sieve_primes1(n)

```
candidates←list [1,2,3….n]
primes←empty list
for i from 2 to n:
   if candidates[i] > 0
      p = candidates[i]
      primes←primes + p
      for j=i+p to n step p:
         candidates[j] = 0
return primes
```

The total number of steps:

$n(1/2 + 1/3 + 1/5 + \ldots + 1/n)$

$1/2 + 1/3 + 1/5 + \ldots + 1/n <$ **$1/2 + 1/3 + 1/4 + 1/5 + \ldots + 1/n = O(\log n)$**

This is the sum of harmonic series: LINK

Sieve of Eratosthenes: upper bound

Algorithm sieve_primes1(n)

```
candidates←list [1,2,3….n]
primes←empty list
for i from 2 to n:
    if candidates[i] > 0
        p = candidates[i]
        primes←primes + p
        for j=i+p to n step p:
            candidates[j] = 0
return primes
```

The total number of steps:

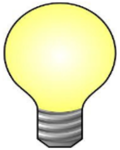
$n + n(1/2 + 1/3 + 1/5 + \ldots + 1/n) = O(n) + O(n \log n) =$ **$O(n \log n)$**

We could make this bound even tighter: O(n log log n)

Algorithmic thinking:
idea makes all the difference

Problem Statement → Algorithm

Naive



Look at the problem from different angles

Eureka!

Good algorithm!

Requirements to your algorithmic solution

Correctness

Speed (fast enough)

Simplicity and **elegance**

Algorithms ...

Quote by *Francis Sullivan*:
"For me, great algorithms are the poetry of computation. Just like verse, they can be terse, allusive, dense, and even mysterious. But once unlocked they cast a brilliant new light on some aspect of computing."