As you join, please write your full name in the chat

Introduction to Algorithms
Science Honors Program (SHP)
Session 8

Christian Lim

Saturday, April 20, 2024

Overview

- **Factoring numbers**
- Break #1
- **Sieve of Eratosthenes**
- Break #2
- **Binary Exponentiation**
- Break #3
- **Euclidean Algorithm for computing greatest common divisor (gcd)**

Refresher of efficiency analysis

```
sum = 0

for i from 1 to n:
  sum = sum + i

output sum
```

$O(n)$

Now your turn

```
sum = 0

for i from 1 to n:
    for j from 1 to 2*n:
        sum = sum + i * j

output sum
```

$O(n^2)$

Factoring numbers

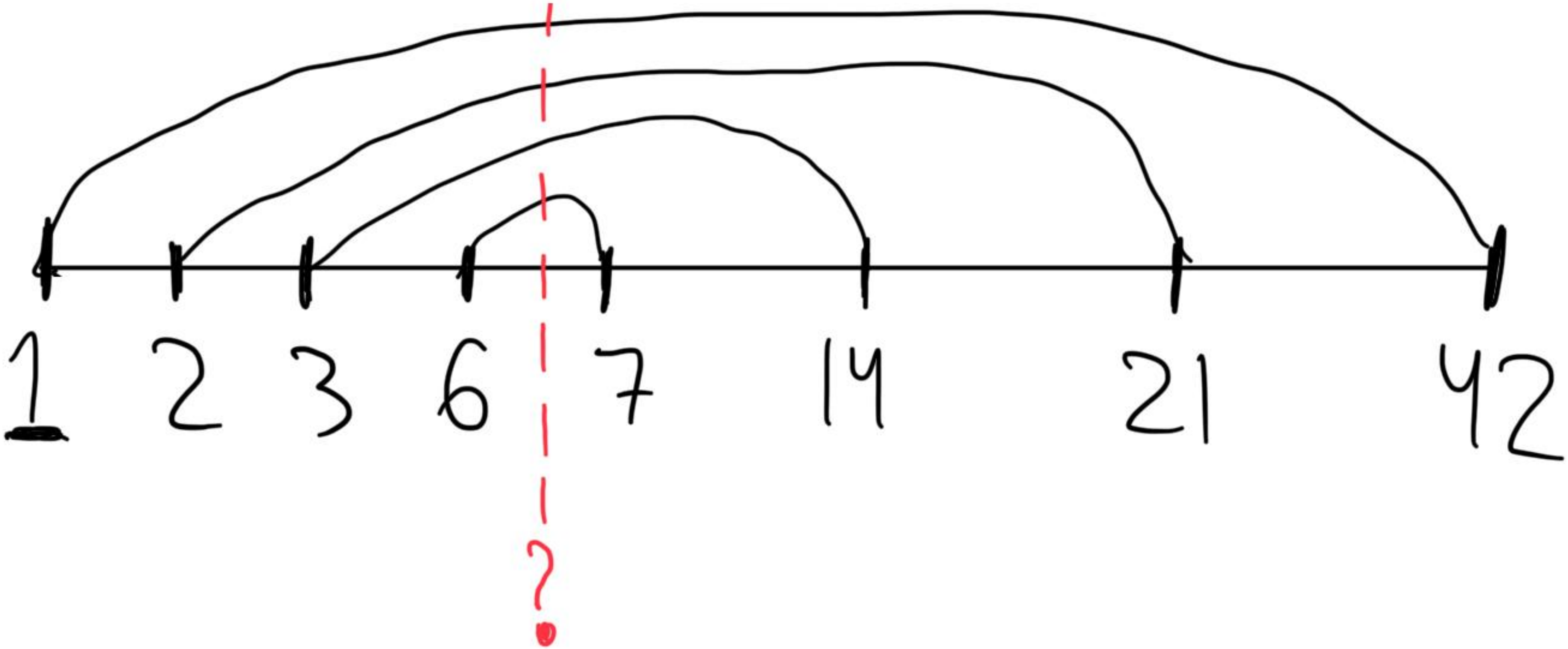
- For example:
  Divisors of 42 are 1, 2, 3, 6, 7, 14, 21, 42.

```
for d from 1 to n:
  if n is divisible by d:
    record that d is a divisor
```

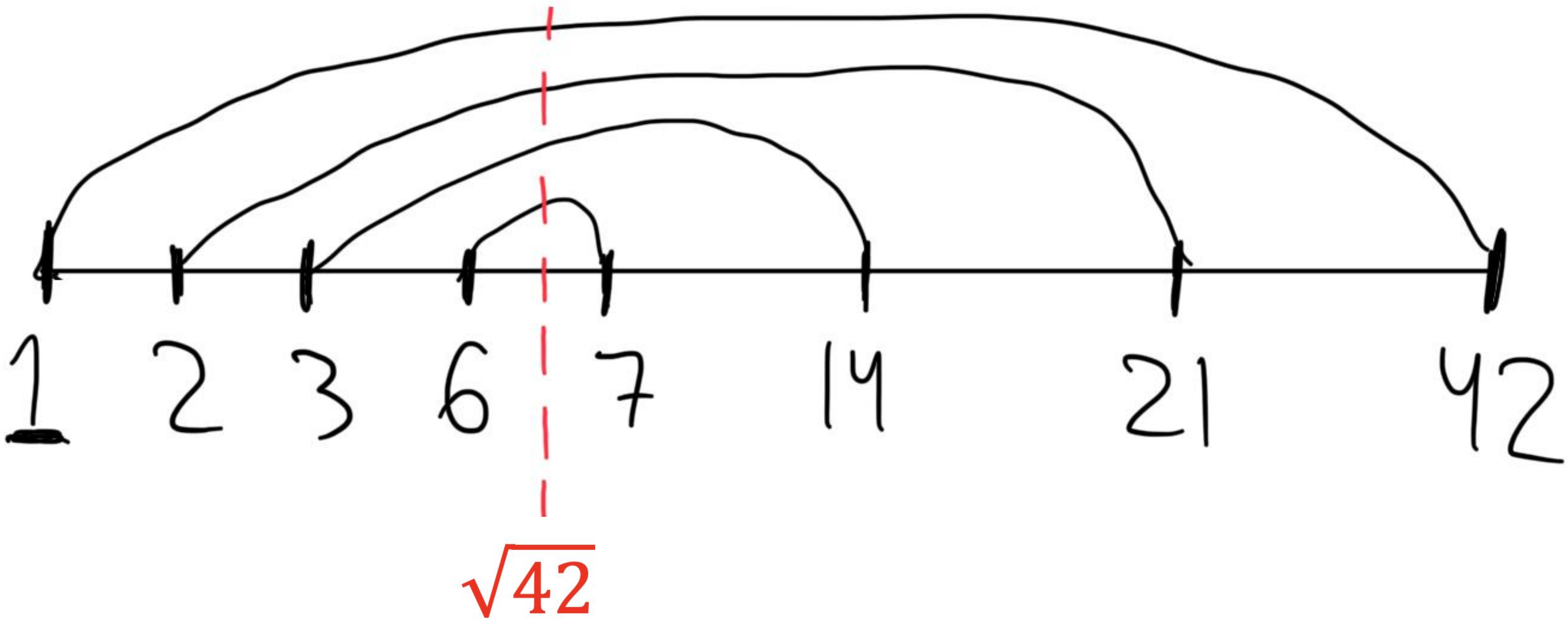
n%d==0

O(n)

Can we do better?



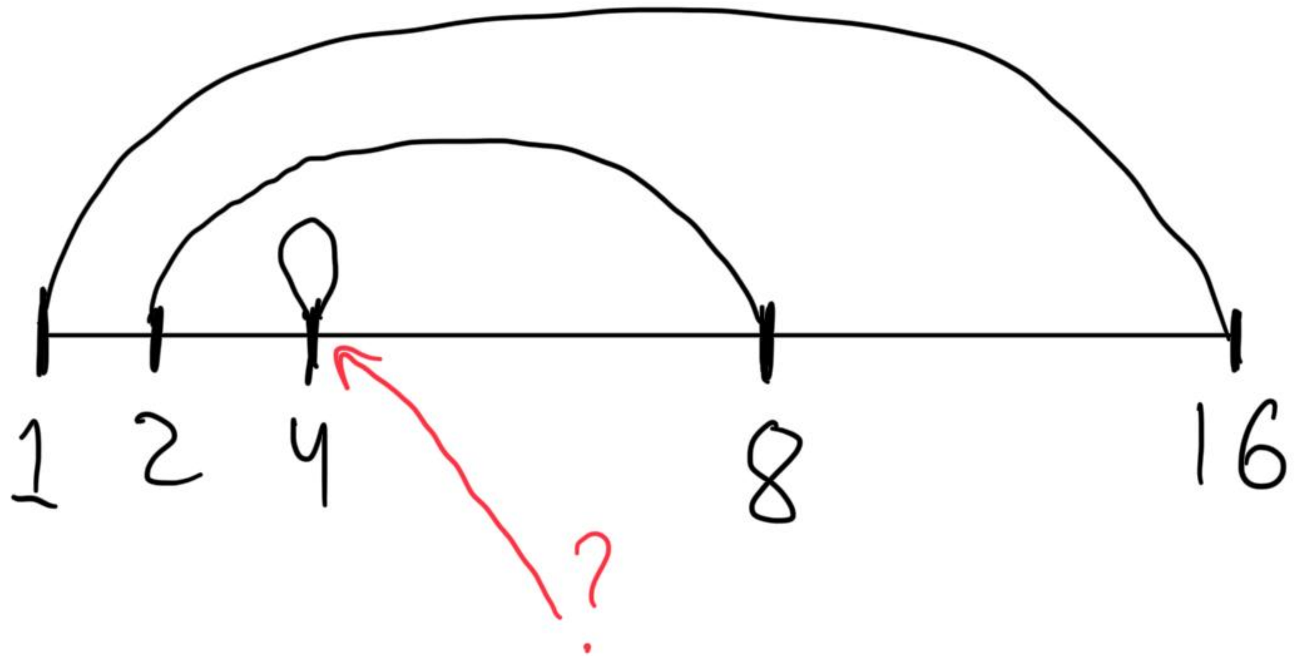
Can we do better?



Main idea

- Consider a pair of divisors (x,y) such that $x*y=n$.
    - **Claim:** either x or y must be smaller than or equal to $\sqrt{n}$.
    - If both are bigger than square root, then their product is bigger than n.
- Then, each pair can be enumerated by the smaller of the two divisors.

Gotta be careful



The algorithm and its efficiency

```
for d from 1 to √n:
  if n % d == 0:
    record that d is a divisor
    if d != n // d:
      record that n // d is a divisor
```

$O(\sqrt{n})$

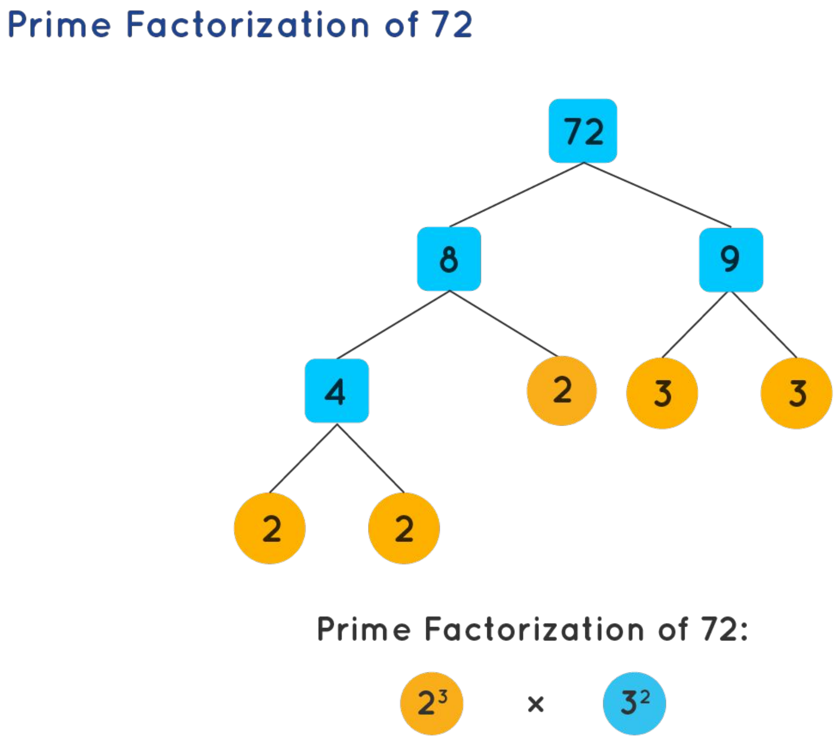
Why does this matter?

n=10:
O(n) ~ 10 operations
O(√n) ~ 3 operations

n=1000:
O(n) ~ 1000 operations
O(√n) ~ 30 operations

n = a million:
O(n) ~ a million operations
O(√n) ~ a thousand operations

Prime Factorization



A simple prime factorization algorithm

```
for x from 2 to n:
    divide n by x while you can
```

(every time that you divide by *x*, record that it's a factor)

Example:

n=28=2*2*7

2 divides $n=28$: now n is 14 and 2 is a factor.
2 divides $n=14$: now n is 7 and 2 is a factor.
3 doesn't divide $n=7$.
4 doesn't divide $n=7$.
5 doesn't divide $n=7$.
6 doesn't divide $n=7$.
7 divides $n=7$: now n is 1 and 7 is a factor.

$O(n)$

How can we optimize it?

The algorithm and its efficiency

```
current = n
for x from 2 to √n:
  while current % x == 0:
    current = current // x
    record that x is a prime factor

if current > 1:
  record that current is a prime factor
```

$O(\sqrt{n})$

Practice Problem #1

- https://leetcode.com/problems/four-divisors/description/

BREAK #1

A refresher

Using the algorithms just learned, how fast can we check whether or not a number is prime?

$$O(\sqrt{n})$$

How about finding all primes up to *n*?

Sieve of Eratosthenes: Main Idea

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

2	3	~~4~~	5	~~6~~	7	~~8~~	9	~~10~~	11	~~12~~	13	~~14~~	15	~~16~~

2	**3**	~~4~~	5	~~6~~	7	~~8~~	~~9~~	~~10~~	11	~~12~~	13	~~14~~	~~15~~	~~16~~

2	**3**	~~4~~	**5**	~~6~~	7	~~8~~	~~9~~	~~10~~	11	~~12~~	13	~~14~~	~~15~~	~~16~~

2	**3**	~~4~~	**5**	~~6~~	**7**	~~8~~	~~9~~	~~10~~	11	~~12~~	13	~~14~~	~~15~~	~~16~~

2	**3**	~~4~~	**5**	~~6~~	**7**	~~8~~	~~9~~	~~10~~	**11**	~~12~~	13	~~14~~	~~15~~	~~16~~

2	**3**	~~4~~	**5**	~~6~~	**7**	~~8~~	~~9~~	~~10~~	**11**	~~12~~	**13**	~~14~~	~~15~~	~~16~~

Sieve of Eratosthenes: Visualization

Sieve of Eratosthenes: algorithm

```
initialize an array is_prime of size n+1,
  storing True for each number

for i from 2 to n:
  if is_prime[i]:
    for each multiple j ≤ n:
      is_prime[j] = False
```

Python implementation

```
for i in range(2, n + 1):
    if is_prime[i]:
        for j in range(i * 2, n + 1, i):
            is_prime[j] = False
```

Efficiency? At least an upper bound

```
for i in range(2, n + 1):
  if is_prime[i]:
    for j in range(i * 2, n + 1, i):
      is_prime[j] = False
```

- For each *i*, we perform around *n/i* operations.
  - Because $j = 2*i, 3*i, \dots, (n/i)*i$.
- Then, our algorithm performs no more than:
  - $n/2 + n/3 + n/4 + n/5 + n/6 + \dots + n/n$
  - operations
- How to estimate this sum?!

We are trying to estimate:

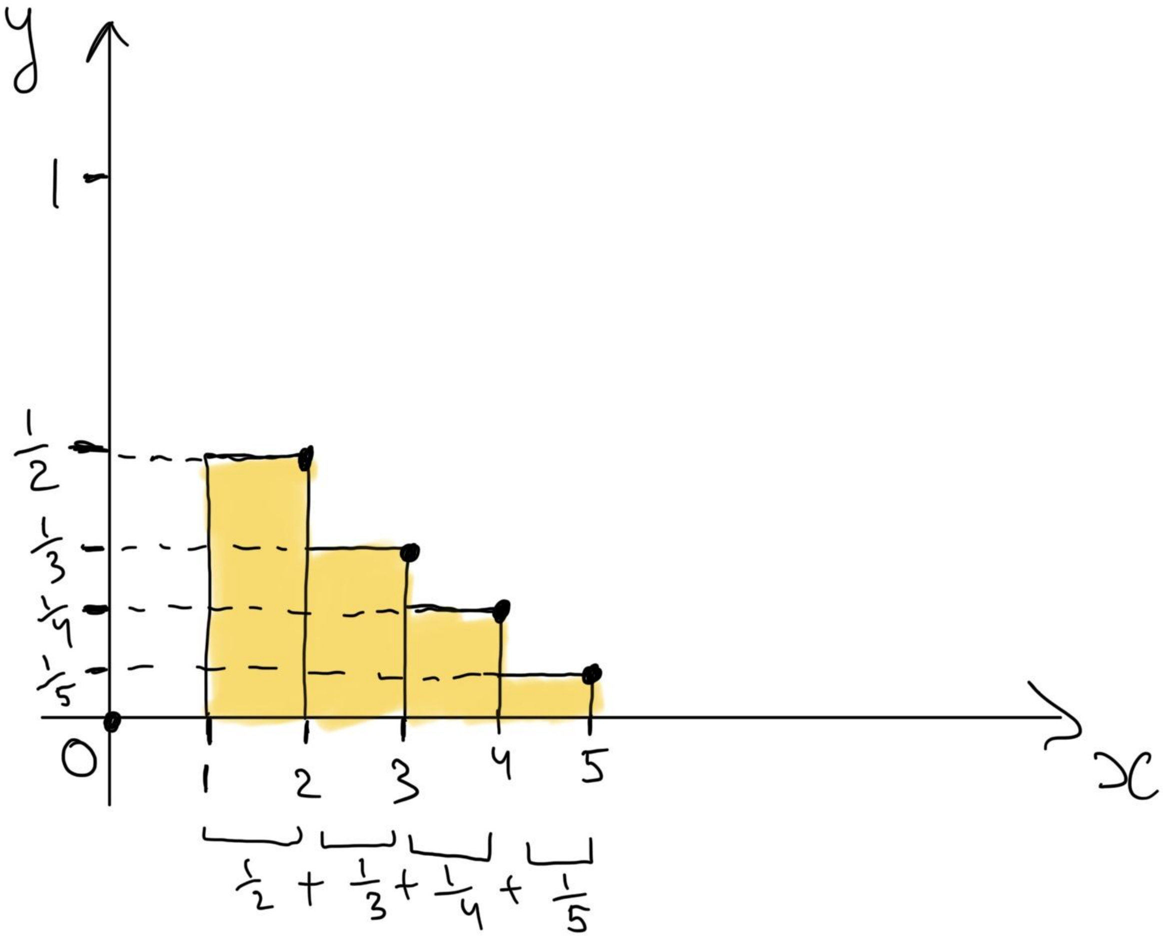
$$\frac{n}{2}+\frac{n}{3}+\cdots+\frac{n}{n}=n\left(\frac{1}{2}+\frac{1}{3}+\cdots+\frac{1}{n}\right)$$

Let's estimate the parenthesized sum:

$$\frac{1}{2}+\frac{1}{3}+\cdots+\frac{1}{n}$$

For n=5

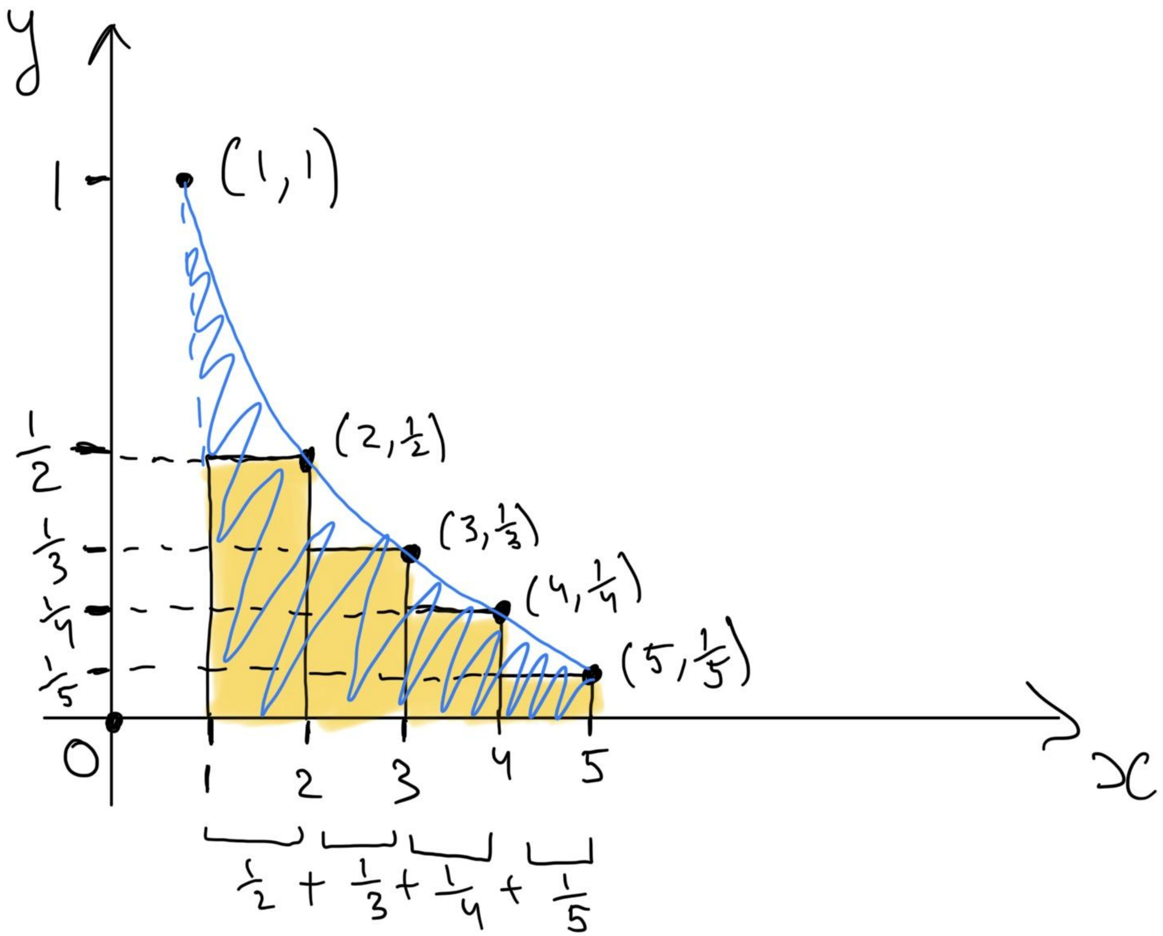
$$\frac{1}{2}+\frac{1}{3}+\cdots+\frac{1}{n}$$



For n=5

$$\frac{1}{2}+\frac{1}{3}+\cdots+\frac{1}{n}\leq \text{Blue area}$$

How to find the blue area?



Calculus!

$$\text{Blue area} = \int_1^n \frac{1}{x}\,dx = [\ln x]_1^n = \ln n$$

So,

$$\frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} \le \ln n$$

$$n\left(\frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}\right)$$

So, our algorithm does at most $n \ln n$ steps.

Efficiency of the Sieve of Eratosthenes

$O(n \log n)$ as an upper bound.

A naive primality check for each number would be $O(n\sqrt{n})$.

For $n=10^9$:

$O(n \log n) \sim 2*10^{10}$ operations
$O(n\sqrt{n}) \sim 3*10^{13}$ operations

Fun fact: the true efficiency of the sieve is $O(n \log (\log n))$, but it's kinda hard to show :)

Practice Problem #2

- https://leetcode.com/problems/count-primes/description/

Practice Problem #3

- https://codeforces.com/contest/26/problem/A

BREAK #2

Raising a number to a power

How would you compute 5^4 in your head?

- If you've multiplied 5 by itself 4 times, you are boring.
- If you've computed 5^2 and squared it, you are thinking algorithmically!
- If you've memorized it, you are too cool.

Naive algorithm

To compute a^n, let's just multiply a by itself a bunch of times:

```
result = 1
for i in range(n):
  result *= a  # the same as: result = result * a
```

$O(n)$

Binary Exponentiation: Main Idea

$$a^n = \begin{cases} 1 & \text{if } n == 0 \\ \left(a^{\frac{n}{2}}\right)^2 & \text{if } n > 0 \text{ and } n \text{ even} \\ \left(a^{\frac{n-1}{2}}\right)^2 \cdot a & \text{if } n > 0 \text{ and } n \text{ odd} \end{cases}$$

Binary Exponentiation: Implementation

```
function power(a, n):
  if n == 0:
    return 1
  elif n % 2 == 0:
    result = power(a, n // 2)
    return result * result
  else:
    result = power(a, (n - 1) // 2)
    return result * result * a
```

Example

power(2, 18) → power(2, 9) → power(2, 4) → power(2, 2) → power(2, 1) → power(2, 0)

*512*512 = **262144** ← 16*16*2 = **512** ← 4*4 = **16** ← 2*2 = **4** ← 1*1*2 = **2** ← **1***

```
function power(a, n):
  if n == 0:
    return 1
  elif n % 2 == 0:
    result = power(a, n // 2)
    return result * result
  else:
    result = power(a, (n - 1) // 2)
    return result * result * a
```

Let’s talk efficiency

On every step, we reduce our exponent by a factor of 2.

So, if we’ve done k operations, we’ve reduced the exponent to $n/2^k$.

We stop when the exponent is 1. At that point, $n/2^k = 1$.

How many operations do we perform in total?

$2^k = n$, so $k = \log_2(n)$.

So, the algorithm has complexity ***O(log n)***.

Naive vs Binary Exponentiation

For $n=1000$:

$O(n)$ ~ 1000 operations
$O(\log n)$ ~ 10 operations

Practice Problem #4

- https://leetcode.com/problems/count-good-numbers/description/

We’ll end here :)

Columbia University Local Contest (CULC)

- We will have 3rd **Columbia University Local Contest** (CULC)
  - Individual, not team, contest!
  - Date: **Saturday, April 27, 2024**
  - Time: **2pm ET ~ 7pm ET**
  - @**Uris Hall**
- **https://bit.ly/spring2024-culc-flyer**

- Please sign up using: **https://bit.ly/icpc-culc-registration**

Summer 2024 Coaching

- **Christian Lim** can help you getting better in programming contests throughout the summer and beyond via a small team-based coaching.

- https://bit.ly/christian-lim-coaching

THANK YOU

BREAK #3

Euclidean Algorithm

- Given two non-negative integers *a* and *b*, we have to find their GCD (greatest common divisor), i.e. the largest number which is a divisor of both a and `b`. It's commonly denoted by `gcd(a,b)`.
- For example, *gcd(12, 16) = 4*; *gcd(5, 15) = 5; gcd(3, 0) = 3*.

Observations

- $gcd(a, b) = gcd(b, a)$
- $gcd(a \pm b, b) = gcd(a, b)$
- $gcd(a, b \pm a) = gcd(a, b)$

Why?

Let's show that $\boldsymbol{gcd(a + b, b) = gcd(a, b)}$. Let $g = gcd(a, b)$.

g divides a and g divides b, so g divides $a+b$. So $gcd(a + b, b)$ is at least g.

Suppose $gcd(a + b, b)=g^*$, where $g^* > g$. Since g^* divides b and $(a+b)$, g^* must also divide (a+b)-b = a.

Since g^* divides both a and b, so $gcd(a, b)$ is at least g^*. But we know that it is g, which is less than g^*. **Contradiction.**

Algorithm idea

Suppose $a>b$. Let's subtract b from a many times until a becomes less than b. Then, let's swap a and b and continue the same procedure. We know that the GCD of a and b **does not change** due to these transformations.

Notice that the numbers keep getting smaller and smaller.

At some point, we'll have $a=g$ and $b=0$. At this point, we know that GCD is g.

Let's formalize it

What does it mean "to subtract *b* from *a* until *a* becomes less than *b*"?

This is the same as just replacing *a* with (*a mod b*): the remainder of *a* when divided by *b*.

So, we have an algorithm.

Euclidean Algorithm: Outline

$$\gcd(a, b) = \begin{cases} a, & \text{if } b = 0 \\ \gcd(b, a \bmod b), & \text{otherwise.} \end{cases}$$

Euclidean Algorithm: Implementation

```python
def gcd(a, b):
  if b == 0:
    return a
  else:
    return gcd(b, a % b)
```

Analysis of efficiency

```python
def gcd(a, b):
    if b == 0:
        return a
    else:
        return gcd(b, a % b)
```

Properties of **%** operation:

- $(a \% b) < b$
- For $a \geq b$, $(a \% b) < \frac{1}{2} a$

Notice:

- In our algorithm, $a \geq b$ always because $b > (a\%b)$.
- Hence, $(a \% b) < \frac{1}{2} a$.
- Thus, on each step, we reduce one of the numbers by a factor of 2 at least.
- By a similar analysis as before, we can perform at most $\boldsymbol{\log_2(a) + \log_2(b)}$ operations in total.
- Hence, efficiency is $\boldsymbol{O(\log(a) + \log(b))}$.