CS 350 Algorithms and Complexity

Winter 2019

Lecture 3: Analyzing Non-Recursive Algorithms

Andrew P. Black

Department of Computer Science

Portland State University

Analysis of time efficiency

- Time efficiency is analyzed by determining the number of repetitions of the "basic operation"
- Almost always depends on the size of the input
- "Basic operation": the operation that contributes most towards the running time of the algorithm

$$T(n) \approx c_{op} \times C(n)$$

Analysis of time efficiency

- Time efficiency is analyzed by determining the number of repetitions of the "basic operation"
- Almost always depends on the size of the input
- "Basic operation": the operation that contributes most towards the running time of the algorithm

run time

$$T(n) \approx c_{op} \times C(n)$$

Analysis of time efficiency

- Time efficiency is analyzed by determining the number of repetitions of the "basic operation"
- Almost always depends on the size of the input
- "Basic operation": the operation that contributes most towards the running time of the algorithm

cost of basic op: constant

run time

$$T(n) \approx c_{op} \times C(n)$$

Analysis of time efficiency

- Time efficiency is analyzed by determining the number of repetitions of the "basic operation"
- Almost always depends on the size of the input
- "Basic operation": the operation that contributes most towards the running time of the algorithm

$$T(n) \approx c_{op} \times C(n)$$

run time

cost of basic op: constant

number of times basic op is executed

Problem	Input size measure	Basic operation
Searching for key in a list of *n* items		
Multiplication of two matrices		
Checking primality of a given integer *n*		
Shortest path through a graph		

Complete the table

Problem	Input size measure	Basic operation
Searching for key in a list of *n* items	A: Number of list's items, i.e. n	A: Key comparison
Multiplication of two matrices	B: Matrix dimension, or total number of elements	B: Multiplication of two numbers
Checking primality of a given integer *n*	C: size of n = number of digits	C: Division
Shortest path through a graph	D: #vertices and/or edges	D: Visiting a vertex or traversing an edge

Problem	Input size measure	Basic operation
Searching for key in a list of n items	A: Number of list's items, i.e. n	
	B: Matrix dimension, or total number of elements	
	C: size of n = number of digits	
	D: #vertices and/or edges	

Problem	Input size measure	Basic operation
Searching for key in a list of n items		A: Key comparison
		B: Multiplication of two numbers
		C: Division
		D: Visiting a vertex or traversing an edge

Problem	Input size measure	Basic operation
Multiplication of two matrices	A: Number of list's items, i.e. n B: Matrix dimension, or total number of elements C: size of n = number of digits D: #vertices and/or edges	

Problem	Input size measure	Basic operation
Multiplication of two matrices		A: Key comparison B: Multiplication of two numbers C: Division D: Visiting a vertex or traversing an edge

Problem	Input size measure	Basic operation
	A: Number of list's items, i.e. n	
	B: Matrix dimension, or total number of elements	
Checking primality of a given integer *n*	C: size of n = number of digits	
	D: #vertices and/or edges	

Problem	Input size measure	Basic operation
Checking primality of a given integer *n*		A: Key comparison B: Multiplication of two numbers C: Division D: Visiting a vertex or traversing an edge

Problem	Input size measure	Basic operation
	A: Number of list's items, i.e. n	
	B: Matrix dimension, or total number of elements	
	C: size of n = number of digits	
Shortest path through a graph	D: #vertices and/or edges	

Problem	Input size measure	Basic operation
		A: Key comparison
		B: Multiplication of two numbers
		C: Division
Shortest path through a graph		D: Visiting a vertex or traversing an edge

Problem	Input size measure	Basic operation
Searching for key in a list of *n* items		
Multiplication of two matrices		
Checking primality of a given integer *n*		
Shortest path through a graph		

Best-case, average-case, worst-case

- For some algorithms, efficiency depends on the input:
- Worst case: $C_{worst}(n)$ – maximum over inputs of size n
- Best case: $C_{best}(n)$ – minimum over inputs of size n
- Average case: $C_{avg}(n)$ – "average" over inputs of size n
  - Number of times the basic operation will be executed on typical input
    - Not the average of worst and best case
  - Expected number of basic operations under some assumption about the probability distribution of all possible inputs

Discuss:

ALGORITHM *UniqueElements*$(A[0..n-1])$
//Determines whether all the elements in a given array are distinct
//Input: An array $A[0..n-1]$
//Output: Returns "true" if all the elements in A are distinct
//   and "false" otherwise
for $i \leftarrow 0$ **to** $n-2$ **do**
    for $j \leftarrow i+1$ **to** $n-1$ **do**
        if $A[i] = A[j]$ **return false**
return true

- What's the best case, and its running time?
  A. constant — $O(1)$
  B. linear — $O(n)$
  C. quadratic — $O(n^2)$

Discuss:

ALGORITHM *UniqueElements*$(A[0..n-1])$
//Determines whether all the elements in a given array are distinct
//Input: An array $A[0..n-1]$
//Output: Returns "true" if all the elements in A are distinct
//   and "false" otherwise
for $i \leftarrow 0$ **to** $n-2$ **do**
    for $j \leftarrow i+1$ **to** $n-1$ **do**
        if $A[i] = A[j]$ **return false**
return true

- What's the worst case, and its running time?
  A. constant — $O(1)$
  B. linear — $O(n)$
  C. quadratic — $O(n^2)$

Discuss:

ALGORITHM *UniqueElements*$(A[0..n-1])$

//Determines whether all the elements in a given array are distinct
//Input: An array $A[0..n-1]$
//Output: Returns "true" if all the elements in A are distinct
//   and "false" otherwise

for $i \leftarrow 0$ **to** $n-2$ **do**
    for $j \leftarrow i+1$ **to** $n-1$ **do**
        if $A[i] = A[j]$ **return false**
return true

- What's the average case, and its running time?
  - A. constant — $O(1)$
  - B. linear — $O(n)$
  - C. quadratic — $O(n^2)$

General Plan for Analysis of non-recursive algorithms

1. Decide on parameter n indicating input size
2. Identify algorithm's basic operation
3. Determine worst, average, and best cases for input of size n
4. Set up a <u>sum</u> for the number of times the basic operation is executed
5. Simplify the sum using standard formulae and rules (see Levitin Appendix A)

“Basic Operation”

ALGORITHM $MaxElement(A[0..n-1])$

//Determines the value of the largest element in a given array
//Input: An array $A[0..n-1]$ of real numbers
//Output: The value of the largest element in A
$maxval \leftarrow A[0]$
for $i \leftarrow 1$ **to** $n-1$ **do**
    if $A[i] > maxval$
        $maxval \leftarrow A[i]$
return $maxval$

- Why choose > as the basic operation?
  - Why not $i \leftarrow i + 1$?
  - Or [ ] ?

Same Algorithm:

ALGORITHM *MaxElement* (*A*: *List*)
    // Determines the value of the largest element in the list A
    // Input: a list A of real numbers
    // Output: the value of the largest element of A
    maxval ← *A.first*
    for *each* **in** A **do**
        if *each* > *maxval*
            maxval ← *each*
    return *maxval*

- Why choose > as the basic operation?
  - Why not $i \leftarrow i + 1$?
  - Or [ ] ?

From Algorithm to Formula

- We want a formula for the # of basic ops
- Basic op will normally be in inner loop
- Bounds of **for** loop become bounds of summation
- e.g. **for** i ← l .. h **do:**

  3 basic operations

- $$\sum_{i=l}^{h} 3$$

Works for nested loops too

ALGORITHM *UniqueElements*$(A[0..n-1])$

//Determines whether all the elements in a given array are distinct
//Input: An array $A[0..n-1]$
//Output: Returns "true" if all the elements in A are distinct
//   and "false" otherwise
for $i \leftarrow 0$ **to** $n-2$ **do**
    for $j \leftarrow i+1$ **to** $n-1$ **do**
        if $A[i] = A[j]$ **return false**
return true

Works for nested loops too

ALGORITHM *UniqueElements*$(A[0..n-1])$

//Determines whether all the elements in a given array are distinct
//Input: An array $A[0..n-1]$
//Output: Returns "true" if all the elements in A are distinct
// and "false" otherwise

for $i \leftarrow 0$ **to** $n-2$ **do**
  for $j \leftarrow i+1$ **to** $n-1$ **do**
    if $A[i] = A[j]$ **return false**
return true

$$\sum_{i=0}^{n-2} \left(\qquad \right)$$

Works for nested loops too

ALGORITHM *UniqueElements*$(A[0..n-1])$
//Determines whether all the elements in a given array are distinct
//Input: An array $A[0..n-1]$
//Output: Returns "true" if all the elements in A are distinct
//   and "false" otherwise
for $i \leftarrow 0$ **to** $n-2$ **do**
    for $j \leftarrow i+1$ **to** $n-1$ **do**
        if $A[i] = A[j]$ **return false**
return true

$$\sum_{i=0}^{n-2}\left(\sum_{j=i+1}^{n-1} 1\right)$$

Useful Summation Formulae

$\Sigma_{l \le i \le u} 1 =$

In particular, $\Sigma_{1 \le i \le n} 1 = n$

$\Sigma_{1 \le i \le n}\, i =$

$\Sigma_{1 \le i \le n}\, i^2 =$

$\Sigma_{0 \le i \le n}\, a^i =$

In particular, $\Sigma_{0 \le i \le n} 2^i = 2$

$\Sigma(a_i \pm b_i) =$

$\Sigma c\, a_i =$

$\Sigma_{l \le i \le u}\, a_i =$

Useful Summation Formulae

$\Sigma_{l \leq i \leq u} 1 = 1 + 1 + \ldots + 1 = u - l + 1$

In particular, $\Sigma_{1 \leq i \leq n} 1 = n$

$\Sigma_{1 \leq i \leq n} i =$

$\Sigma_{1 \leq i \leq n} i^2 =$

$\Sigma_{0 \leq i \leq n} a^i =$

In particular, $\Sigma_{0 \leq i \leq n} 2^i = 2$

$\Sigma(a_i \pm b_i) =$

$\Sigma c\, a_i =$

$\Sigma_{l \leq i \leq u} a_i =$

Useful Summation Formulae

$\Sigma_{l \le i \le u} 1 = 1+1+\ldots+1 = u - l + 1$

In particular, $\Sigma_{1 \le i \le n} 1 = n - 1 + 1 = n \in \Theta(n)$

$\Sigma_{1 \le i \le n}\, i =$

$\Sigma_{1 \le i \le n}\, i^2 =$

$\Sigma_{0 \le i \le n}\, a^i \;=$

In particular, $\Sigma_{0 \le i \le n}\, 2^i \;= 2$

$\Sigma(a_i \pm b_i) =$

$\Sigma c\, a_i \;=$

$\Sigma_{l \le i \le u}\, a_i \;=$

Useful Summation Formulae

$\Sigma_{l \le i \le u} 1 = 1+1+\ldots+1 = u - l + 1$

In particular, $\Sigma_{1 \le i \le n} 1 = n - 1 + 1 = n \in \Theta(n)$

$\Sigma_{1 \le i \le n}\, i = 1+2+\ldots+n = n(n+1)/2 \approx n^2/2 \in \Theta(n^2)$

$\Sigma_{1 \le i \le n}\, i^2 =$

$\Sigma_{0 \le i \le n}\, a^i =$

In particular, $\Sigma_{0 \le i \le n} 2^i = 2$

$\Sigma(a_i \pm b_i) =$

$\Sigma c\, a_i =$

$\Sigma_{l \le i \le u}\, a_i =$

Useful Summation Formulae

$\Sigma_{l\leq i\leq u} 1 = 1+1+\ldots+1 = u - l + 1$

In particular, $\Sigma_{1\leq i\leq n} 1 = n - 1 + 1 = n \in \Theta(n)$

$\Sigma_{1\leq i\leq n}\, i = 1+2+\ldots+n = n(n+1)/2 \approx n^2/2 \in \Theta(n^2)$

$\Sigma_{1\leq i\leq n}\, i^2 = 1^2+2^2+\ldots+n^2 = n(n+1)(2n+1)/6 \approx n^3/3 \in \Theta(n^3)$

$\Sigma_{0\leq i\leq n}\, a^i =$

In particular, $\Sigma_{0\leq i\leq n} 2^i = 2$

$\Sigma(a_i \pm b_i) =$

$\Sigma c\, a_i =$

$\Sigma_{l\leq i\leq u}\, a_i =$

Useful Summation Formulae

$\Sigma_{l \le i \le u} 1 = 1+1+\ldots+1 = u - l + 1$

In particular, $\Sigma_{1 \le i \le n} 1 = n - 1 + 1 = n \in \Theta(n)$

$\Sigma_{1 \le i \le n}\, i = 1+2+\ldots+n = n(n+1)/2 \approx n^2/2 \in \Theta(n^2)$

$\Sigma_{1 \le i \le n}\, i^2 = 1^2+2^2+\ldots+n^2 = n(n+1)(2n+1)/6 \approx n^3/3 \in \Theta(n^3)$

$\Sigma_{0 \le i \le n}\, a^i = 1 + a + \ldots + a^n = (a^{n+1} - 1)/(a - 1)$ for any $a \neq 1$

In particular, $\Sigma_{0 \le i \le n} 2^i = 2$

$\Sigma(a_i \pm b_i) =$

$\Sigma c\, a_i =$

$\Sigma_{l \le i \le u}\, a_i =$

Useful Summation Formulae

$\Sigma_{l \le i \le u} 1 = 1+1+\ldots+1 = u - l + 1$

In particular, $\Sigma_{1 \le i \le n} 1 = n - 1 + 1 = n \in \Theta(n)$

$\Sigma_{1 \le i \le n}\, i = 1+2+\ldots+n = n(n+1)/2 \approx n^2/2 \in \Theta(n^2)$

$\Sigma_{1 \le i \le n}\, i^2 = 1^2+2^2+\ldots+n^2 = n(n+1)(2n+1)/6 \approx n^3/3 \in \Theta(n^3)$

$\Sigma_{0 \le i \le n}\, a^i = 1 + a + \ldots + a^n = (a^{n+1} - 1)/(a - 1)$ for any $a \ne 1$

In particular, $\Sigma_{0 \le i \le n} 2^i = 2^0 + 2^1 + \ldots + 2^n = 2^{n+1} - 1 \in \Theta(2^n)$

$\Sigma(a_i \pm b_i) =$

$\Sigma c\, a_i =$

$\Sigma_{l \le i \le u}\, a_i =$

Useful Summation Formulae

$\Sigma_{l \le i \le u} 1 = 1+1+\ldots+1 = u - l + 1$

In particular, $\Sigma_{1 \le i \le n} 1 = n - 1 + 1 = n \in \Theta(n)$

$\Sigma_{1 \le i \le n}\, i = 1+2+\ldots+n = n(n+1)/2 \approx n^2/2 \in \Theta(n^2)$

$\Sigma_{1 \le i \le n}\, i^2 = 1^2+2^2+\ldots+n^2 = n(n+1)(2n+1)/6 \approx n^3/3 \in \Theta(n^3)$

$\Sigma_{0 \le i \le n}\, a^i = 1 + a + \ldots + a^n = (a^{n+1} - 1)/(a - 1)$ for any $a \neq 1$

In particular, $\Sigma_{0 \le i \le n} 2^i = 2^0 + 2^1 + \ldots + 2^n = 2^{n+1} - 1 \in \Theta(2^n)$

$\Sigma(a_i \pm b_i) = \Sigma a_i \pm \Sigma b_i$

$\Sigma c\, a_i =$

$\Sigma_{l \le i \le u}\, a_i =$

Useful Summation Formulae

$\Sigma_{l \le i \le u} 1 = 1+1+\ldots+1 = u - l + 1$

In particular, $\Sigma_{1 \le i \le n} 1 = n - 1 + 1 = n \in \Theta(n)$

$\Sigma_{1 \le i \le n}\, i = 1+2+\ldots+n = n(n+1)/2 \approx n^2/2 \in \Theta(n^2)$

$\Sigma_{1 \le i \le n}\, i^2 = 1^2+2^2+\ldots+n^2 = n(n+1)(2n+1)/6 \approx n^3/3 \in \Theta(n^3)$

$\Sigma_{0 \le i \le n}\, a^i = 1 + a + \ldots + a^n = (a^{n+1} - 1)/(a - 1)$ for any $a \neq 1$

In particular, $\Sigma_{0 \le i \le n}\, 2^i = 2^0 + 2^1 + \ldots + 2^n = 2^{n+1} - 1 \in \Theta(2^n)$

$\Sigma(a_i \pm b_i) = \Sigma\, a_i \pm \Sigma\, b_i$ $\qquad$ $\Sigma c\, a_i = c\,\Sigma a_i$

$\Sigma_{l \le i \le u}\, a_i =$

Useful Summation Formulae

$\Sigma_{l \le i \le u} 1 = 1+1+\ldots+1 = u - l + 1$

In particular, $\Sigma_{1 \le i \le n} 1 = n - 1 + 1 = n \in \Theta(n)$

$\Sigma_{1 \le i \le n}\, i = 1+2+\ldots+n = n(n+1)/2 \approx n^2/2 \in \Theta(n^2)$

$\Sigma_{1 \le i \le n}\, i^2 = 1^2+2^2+\ldots+n^2 = n(n+1)(2n+1)/6 \approx n^3/3 \in \Theta(n^3)$

$\Sigma_{0 \le i \le n}\, a^i = 1 + a + \ldots + a^n = (a^{n+1} - 1)/(a - 1)$ for any $a \neq 1$

In particular, $\Sigma_{0 \le i \le n} 2^i = 2^0 + 2^1 + \ldots + 2^n = 2^{n+1} - 1 \in \Theta(2^n)$

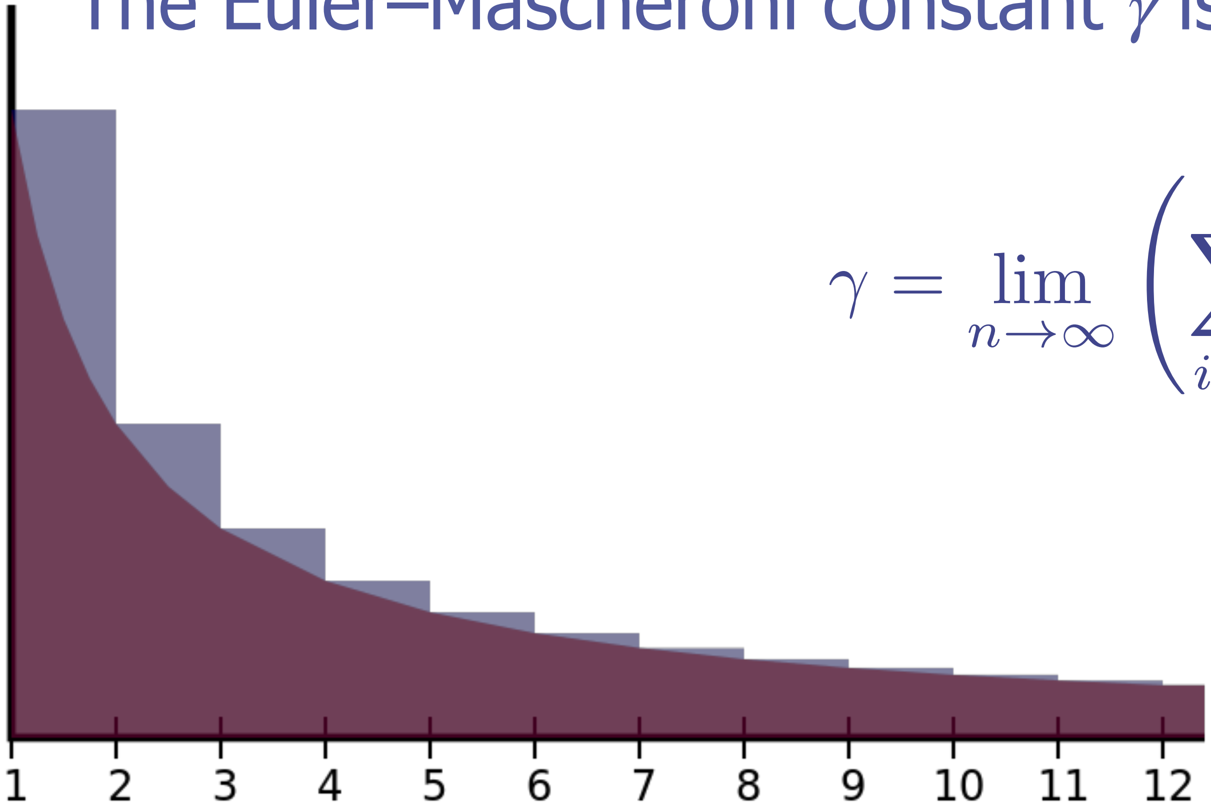
$\Sigma(a_i \pm b_i) = \Sigma a_i \pm \Sigma b_i$ $\qquad$ $\Sigma c\, a_i = c \Sigma a_i$

$\Sigma_{l \le i \le u}\, a_i = \Sigma_{l \le i \le m}\, a_i + \Sigma_{m+1 \le i \le u}\, a_i$

Where do the Summation formulae come from?

- Answer: mathematics.
- Example:

The Euler–Mascheroni constant γ is defined as:

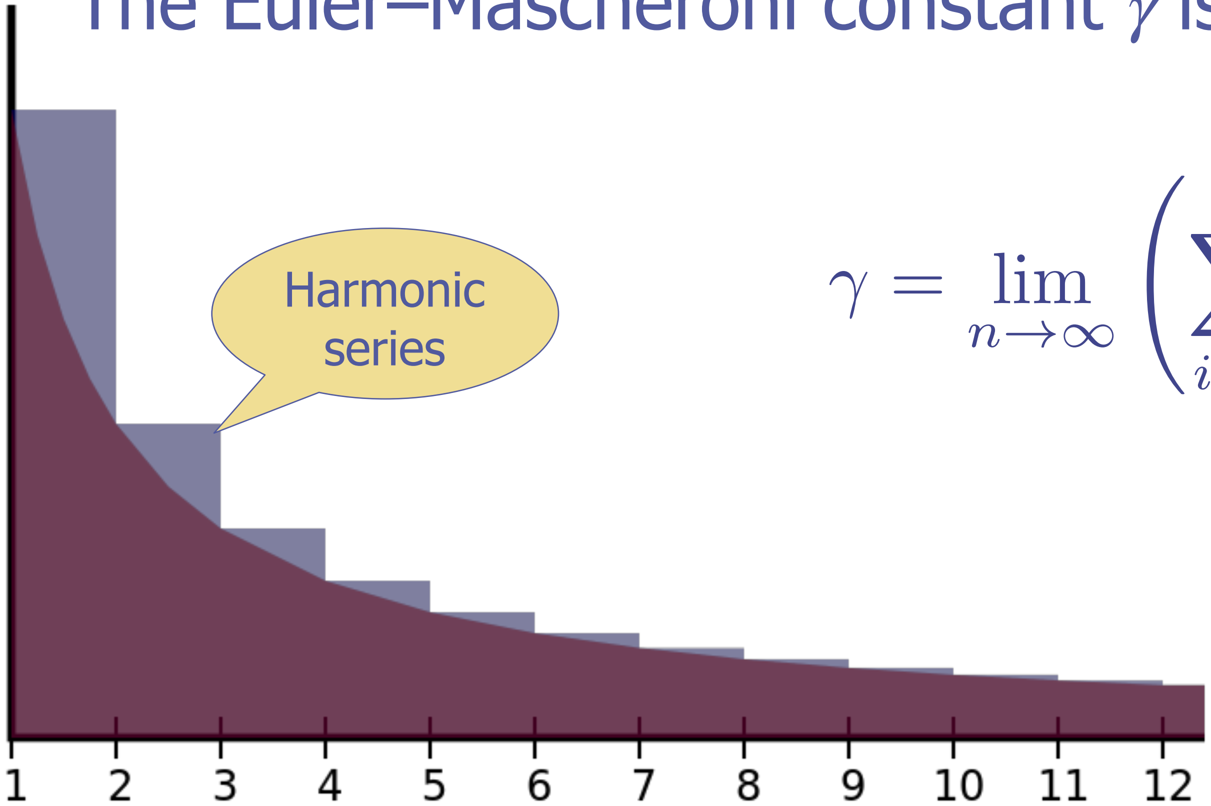


$$\gamma = \lim_{n\to\infty}\left(\sum_{i=1}^{n}\frac{1}{i} - \ln n\right)$$

Where do the Summation formulae come from?

- Answer: mathematics.
- Example:

The Euler–Mascheroni constant γ is <u>defined</u> as:



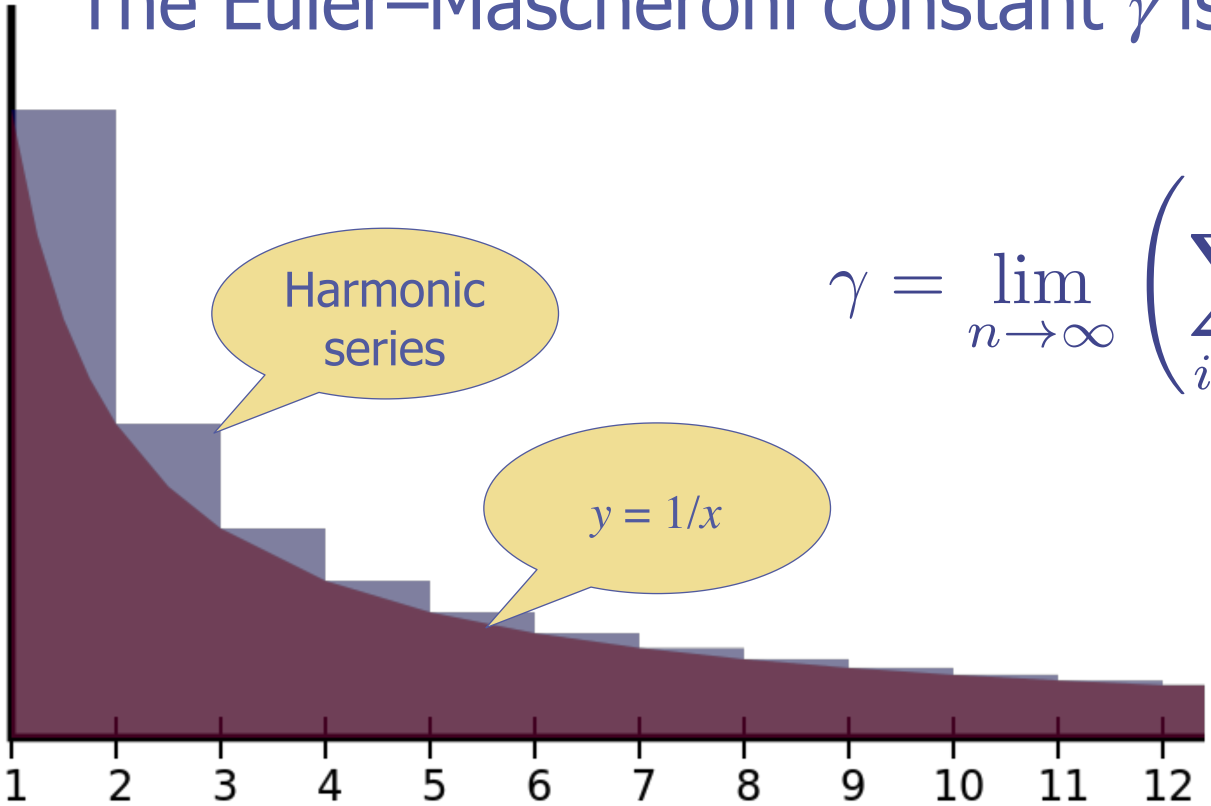
$$\gamma = \lim_{n\to\infty}\left(\sum_{i=1}^{n}\frac{1}{i} - \ln n\right)$$

Where do the Summation formulae come from?

- Answer: mathematics.
- Example:

The Euler–Mascheroni constant γ is <u>defined</u> as:

$$\gamma = \lim_{n\to\infty}\left(\sum_{i=1}^{n}\frac{1}{i} - \ln n\right)$$

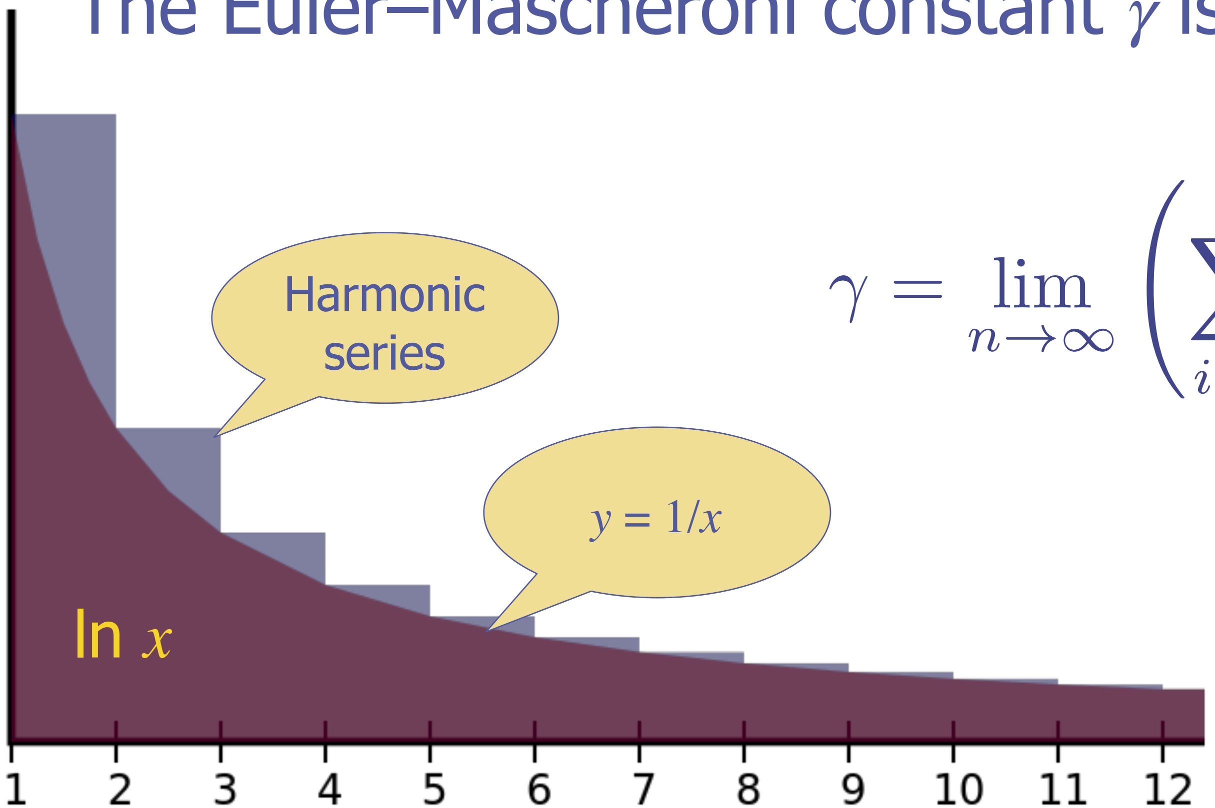


Where do the Summation formulae come from?

- Answer: mathematics.
- Example:

The Euler–Mascheroni constant γ is defined as:

$$\gamma = \lim_{n\to\infty}\left(\sum_{i=1}^{n}\frac{1}{i} - \ln n\right)$$



What does Levitin's $\approx$ mean?

- "becomes almost equal to as $n \to \infty$"
- So formula 8
  $$\sum_{i=1}^{n} \lg i \approx n \lg n$$
  - means

$$\lim_{n\to\infty} \left(\sum_{i=1}^{n} \lg i - n \lg n \right) = 0$$

Example: Counting Binary Digits

ALGORITHM *Binary*(n)

//Input: A positive decimal integer n
//Output: The number of binary digits in n's binary representation
$count \leftarrow 1$
while $n > 1$ **do**
    $count \leftarrow count + 1$
    $n \leftarrow \lfloor n/2 \rfloor$
return $count$

Example: Counting Binary Digits

ALGORITHM $Binary(n)$

//Input: A positive decimal integer n
//Output: The number of binary digits in n's binary representation
$count \leftarrow 1$
while $n > 1$ **do**
    $count \leftarrow count + 1$
    $n \leftarrow \lfloor n/2 \rfloor$
return $count$

- How many times is the basic operation executed?

Example: Counting Binary Digits

ALGORITHM *Binary*(n)

//Input: A positive decimal integer n
//Output: The number of binary digits in n's binary representation
$count \leftarrow 1$
while $n > 1$ **do**
    $count \leftarrow count + 1$
    $n \leftarrow \lfloor n/2 \rfloor$
return $count$

- How many times is the basic operation executed?
- Why is this algorithm harder to analyze than the earlier examples?

Ex 2.3, Problem 1

- Working with a *partner*:

1. Compute the following sums.

   a. $1 + 3 + 5 + 7 + \ldots + 999$

   b. $2 + 4 + 8 + 16 + \ldots + 1024$

   c. $\sum_{i=3}^{n+1} 1$

   d. $\sum_{i=3}^{n+1} i$

   e. $\sum_{i=0}^{n-1} i(i+1)$

   f. $\sum_{j=1}^{n} 3^{j+1}$

   g. $\sum_{i=1}^{n} \sum_{j=1}^{n} ij$

   h. $\sum_{i=1}^{n} 1/i(i+1)$

Ex 2.3, Problem 2

2. Find the order of growth of the following sums.

   a. $\sum_{i=0}^{n-1}(i^2+1)^2$ b. $\sum_{i=2}^{n-1}\lg i^2$

   c. $\sum_{i=1}^{n}(i+1)2^{i-1}$ d. $\sum_{i=0}^{n-1}\sum_{j=0}^{i-1}(i+j)$

   Use the $\Theta(g(n))$ notation with the simplest function $g(n)$ possible.

Ex 2.3, Problem 3

3. The sample variance of n measurements $x_1, x_2, ..., x_n$ can be computed as

$$\frac{\sum_{i=1}^{n}(x_i - \bar{x})^2}{n-1} \text{ where } \bar{x} = \frac{\sum_{i=1}^{n} x_i}{n}$$

or

$$\frac{\sum_{i=1}^{n} x_i^2 - (\sum_{i=1}^{n} x_i)^2/n}{n-1}.$$

Find and compare the number of divisions, multiplications, and additions/subtractions (additions and subtractions are usually bunched together) that are required for computing the variance according to each of these formulas.

Ex 2.3, Problem 4

4. Consider the following algorithm.

Algorithm *Mystery*(n)
//Input: A nonnegative integer n
$S \leftarrow 0$
for $i \leftarrow 1$ **to** n **do**
    $S \leftarrow S + i * i$
return S

Ex 2.3, Problem 4

4. Consider the following algorithm.

Algorithm *Mystery*(n)
//Input: A nonnegative integer n
$S \leftarrow 0$
for $i \leftarrow 1$ **to** n **do**
    $S \leftarrow S + i * i$
return S

What does this algorithm compute?

A. n^2

B. $\sum_{i=1}^{n} i$

C. $\sum_{i=1}^{n} i^2$

D. $\sum_{i=1}^{n} 2i$

Ex 2.3, Problem 4

4. Consider the following algorithm.

Algorithm *Mystery*(n)
//Input: A nonnegative integer n
$S \leftarrow 0$
for $i \leftarrow 1$ **to** n **do**
    $S \leftarrow S + i * i$
return S

Ex 2.3, Problem 4

4. Consider the following algorithm.

Algorithm *Mystery*(n)
//Input: A nonnegative integer n
$S \leftarrow 0$
for $i \leftarrow 1$ **to** n **do**
    $S \leftarrow S + i * i$
return S

What is the basic operation?

A. multiplication

B. addition

C. assignment

D. squaring

Ex 2.3, Problem 4

4. Consider the following algorithm.

Algorithm *Mystery*(n)
//Input: A nonnegative integer n
$S \leftarrow 0$
for $i \leftarrow 1$ **to** n **do**
    $S \leftarrow S + i * i$
return S

How many times is the basic operation executed?

A. once

B. n times

C. $\lg n$ times

D. none of the above

Ex 2.3, Problem 4

4. Consider the following algorithm.

Algorithm *Mystery*(n)
//Input: A nonnegative integer n
$S \leftarrow 0$
for $i \leftarrow 1$ **to** n **do**
    $S \leftarrow S + i * i$
return S

What is the efficiency class of this algorithm? [b is # of bits needed to represent n]

A. $\Theta(1)$

B. $\Theta(n)$

C. $\Theta(b)$

D. $\Theta(2^b)$

Ex 2.3, Problem 4 (cont)

e. Suggest an improvement or a better algorithm altogether and indicate its efficiency class. If you cannot do it, try to prove that, in fact, it cannot be done.

Problem 5 — Group work

5. Consider the following algorithm.

Algorithm $Secret(A[0..n-1])$
//Input: An array $A[0..n-1]$ of n real numbers
$minval \leftarrow A[0]; \quad maxval \leftarrow A[0]$
for $i \leftarrow 1$ **to** $n-1$ **do**
  if $A[i] < minval$
    $minval \leftarrow A[i]$
  if $A[i] > maxval$
    $maxval \leftarrow A[i]$
return $maxval - minval$

a. What does this algorithm compute?
b. What is its basic operation?
c. How many times is the basic operation executed?
d. What is the efficiency class of this algorithm?
e. Suggest an improvement or a better algorithm altogether and indicate its efficiency class.

Ex 2.3, Problem 9

Prove the formula

$$\sum_{i=1}^{n} i = 1 + 2 + ... + n = \frac{n(n+1)}{2}$$

either by mathematical induction or by following the insight of a 10-year old schoolboy named Karl Friedrich Gauss (1777–1855) who grew up to become one of the greatest mathematicians of all times.

Ex 2.3, Problem 11

Algorithm $GE(A[0..n-1, 0..n])$
//Input: An n-by-$n+1$ matrix $A[0..n-1, 0..n]$ of real numbers
for $i \leftarrow 0$ **to** $n-2$ **do**
    for $j \leftarrow i+1$ **to** $n-1$ **do**
        for $k \leftarrow i$ **to** n **do**
            $A[j,k] \leftarrow A[j,k] - A[i,k] * A[j,i] \;/\; A[i,i]$

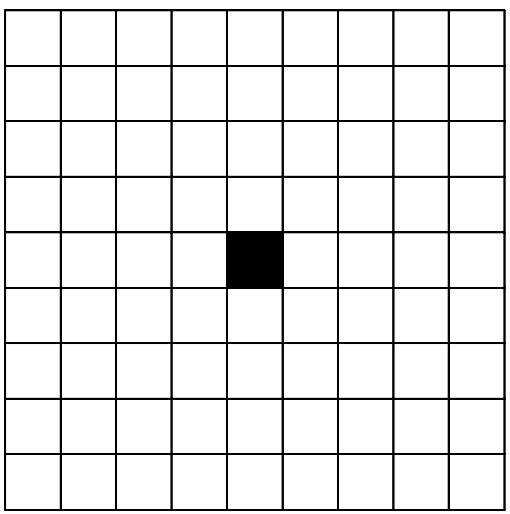
a. Find the time efficiency class of this algorithm

b. What glaring inefficiency does this code contain, and how can it be eliminated?

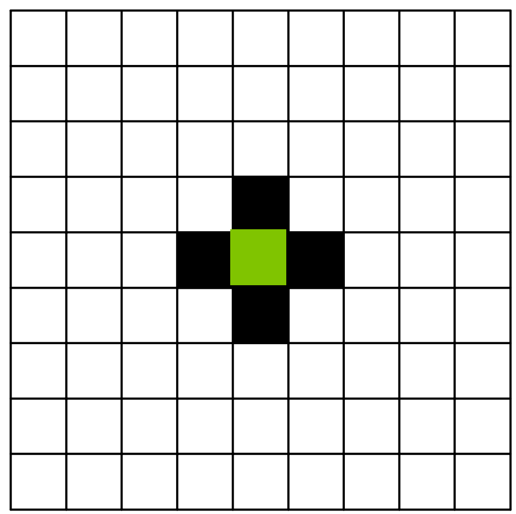
c. Estimate the reduction in run time.

Problem 11: von Neumann neighborhood

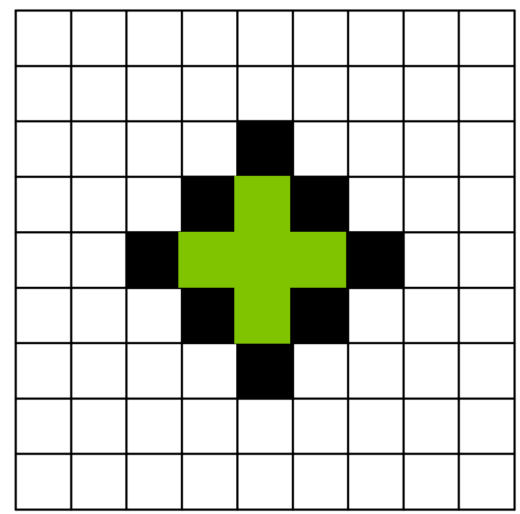
How many one-by-one squares are generated by the algorithm that starts with a single square, and on each of its n iterations adds new squares around the outside. How many one-by-one squares are generated on the n^{th} iteration? Here are the neighborhoods for n = 0, 1, and 2.



$n = 0$



$n = 1$



$n = 2$