Algorithms

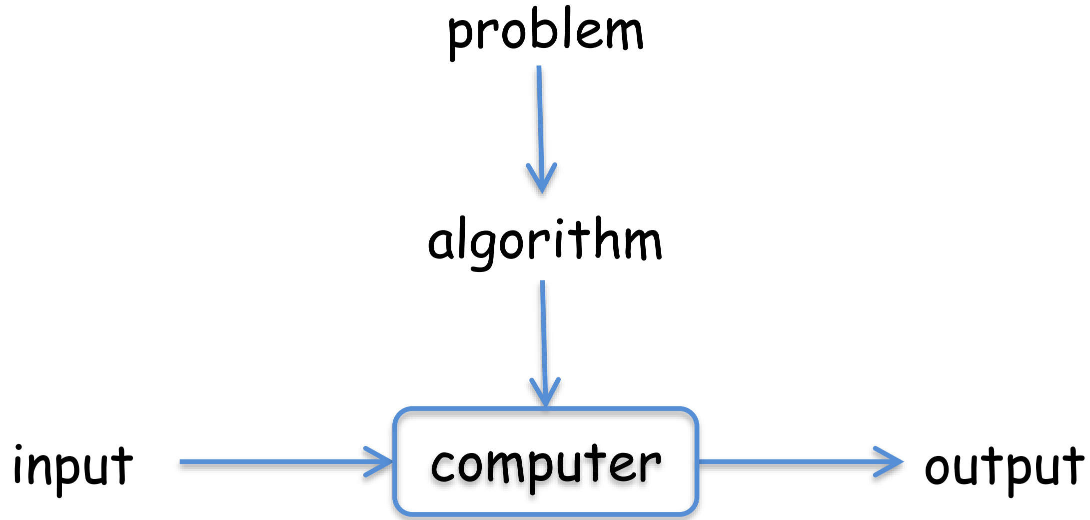
Murat Osmanoglu

Introduction

- 'a sequence of unambiguous instructions for solving a given problem' (Levitin) – obtaining a required output for any legitimate input in a finite amount of time

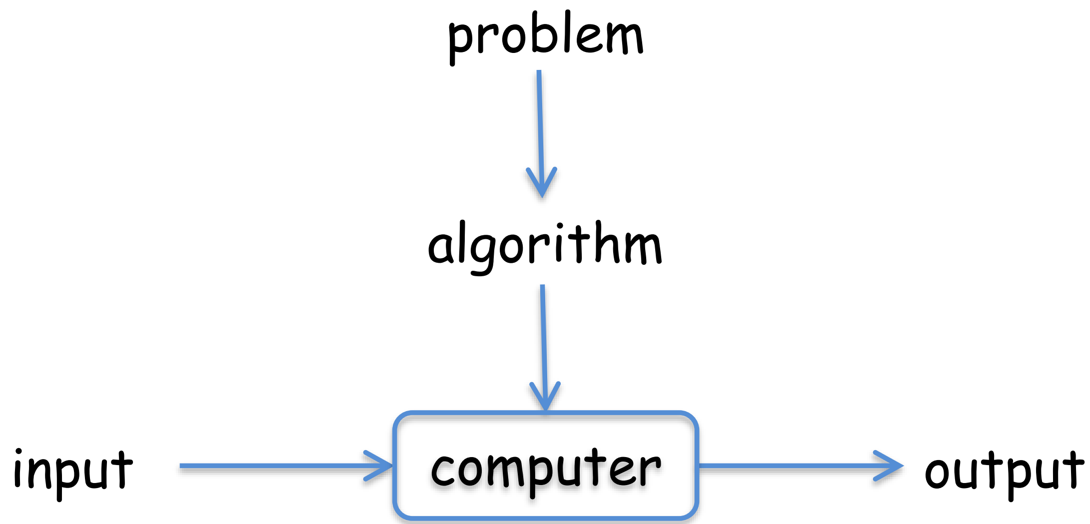
Introduction

- 'a sequence of unambiguous instructions for solving a given problem' (Levitin) – obtaining a required output for any legitimate input in a finite amount of time



Introduction

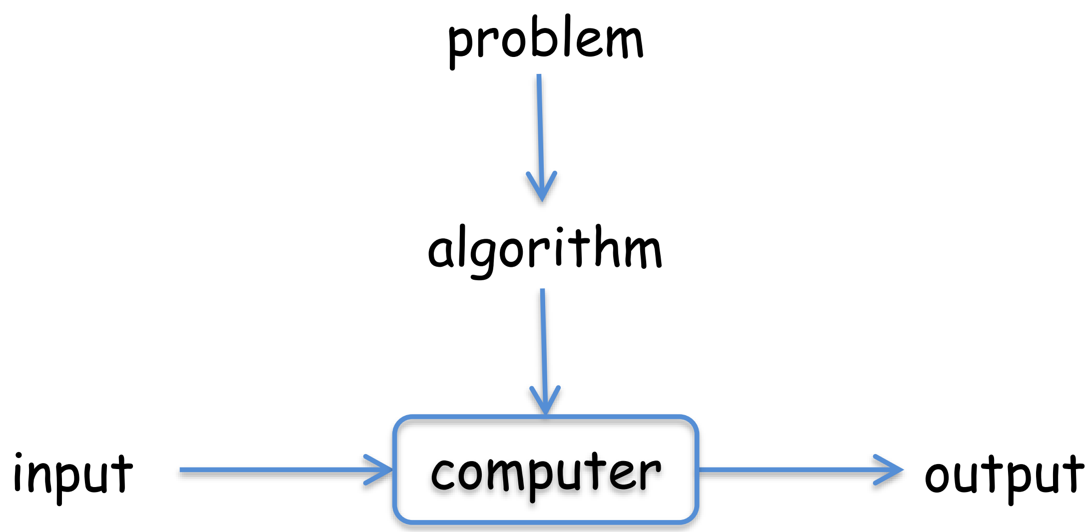
- 'a sequence of unambiguous instructions for solving a given problem' (Levitin) – obtaining a required output for any legitimate input in a finite amount of time



- each step should be expressed in a clear way

Introduction

- 'a sequence of unambiguous instructions for solving a given problem' (Levitin) – obtaining a required output for any legitimate input in a finite amount of time



- each step should be expressed in a clear way
- the nature of the input should be specified carefully

Fundamentals of Algorithmic Problem Solving

Understanding the Problem

Fundamentals of Algorithmic Problem Solving

Understanding the Problem

- design an algorithm to find the maximum number of a finite sequence of integers

Fundamentals of Algorithmic Problem Solving

Understanding the Problem

- design an algorithm to find the maximum number of a finite sequence of integers
  - input : {34, 23, 2, 101, 5, 98, 43}, output : 101

Fundamentals of Algorithmic Problem Solving

Understanding the Problem

- design an algorithm to find the maximum number of a finite sequence of integers
  - input : {34, 23, 2, 101, 5, 98, 43}, output : 101
- an algorithmic problem is specified by describing the set of instances (input) it must work on, and what desired properties the output must have

Fundamentals of Algorithmic Problem Solving

Understanding the Problem

- design an algorithm to find the maximum number of a finite sequence of integers
  - input : {34, 23, 2, 101, 5, 98, 43}, output : 101
- an algorithmic problem is specified by describing the set of instances (input) it must work on, and what desired properties the output must have
- design an algorithm to determine how ‘similar’ two given DNA sequences are

Fundamentals of Algorithmic Problem Solving

Understanding the Problem

- design an algorithm to find the maximum number of a finite sequence of integers
  - input : {34, 23, 2, 101, 5, 98, 43}, output : 101
- an algorithmic problem is specified by describing the set of instances (input) it must work on, and what desired properties the output must have
- design an algorithm to determine how 'similar' two given DNA sequences are
  - DNA sequence is a string of arbitrary length over the alphabet {A, C, G, T},

Fundamentals of Algorithmic Problem Solving

Understanding the Problem

- design an algorithm to find the maximum number of a finite sequence of integers
  - input : {34, 23, 2, 101, 5, 98, 43}, output : 101
- an algorithmic problem is specified by describing the set of instances (input) it must work on, and what desired properties the output must have
- design an algorithm to determine how 'similar' two given DNA sequences are
  - DNA sequence is a string of arbitrary length over the alphabet {A, C, G, T},
  - the degree of the similarity may be measured by the length of their longest common subsequence

Fundamentals of Algorithmic Problem Solving

Understanding the Problem

- design an algorithm to find the maximum number of a finite sequence of integers
  - input : {34, 23, 2, 101, 5, 98, 43}, output : 101
- an algorithmic problem is specified by describing the set of instances (input) it must work on, and what desired properties the output must have
- design an algorithm to determine how 'similar' two given DNA sequences are
  - DNA sequence is a string of arbitrary length over the alphabet {A, C, G, T},
  - the degree of the similarity may be measured by the length of their longest common subsequence

input :

output :

Fundamentals of Algorithmic Problem Solving

Understanding the Problem

- design an algorithm to find the maximum number of a finite sequence of integers
  - input : {34, 23, 2, 101, 5, 98, 43}, output : 101
- an algorithmic problem is specified by describing the set of instances (input) it must work on, and what desired properties the output must have
- design an algorithm to determine how 'similar' two given DNA sequences are
  - DNA sequence is a string of arbitrary length over the alphabet {A, C, G, T},
  - the degree of the similarity may be measured by the length of their longest common subsequence

input : *ACCACTGGT, ACTATCGAG*

output :

Fundamentals of Algorithmic Problem Solving

Understanding the Problem

- design an algorithm to find the maximum number of a finite sequence of integers
  - input : {34, 23, 2, 101, 5, 98, 43}, output : 101
- an algorithmic problem is specified by describing the set of instances (input) it must work on, and what desired properties the output must have
- design an algorithm to determine how 'similar' two given DNA sequences are
  - DNA sequence is a string of arbitrary length over the alphabet {A, C, G, T},
  - the degree of the similarity may be measured by the length of their longest common subsequence

input : ACCACTGGT, ACTATCGAG

output :

Fundamentals of Algorithmic Problem Solving

Understanding the Problem

- design an algorithm to find the maximum number of a finite sequence of integers
  - input : {34, 23, 2, 101, 5, 98, 43}, output : 101
- an algorithmic problem is specified by describing the set of instances (input) it must work on, and what desired properties the output must have
- design an algorithm to determine how 'similar' two given DNA sequences are
  - DNA sequence is a string of arbitrary length over the alphabet {A, C, G, T},
  - the degree of the similarity may be measured by the length of their longest common subsequence

input : ACCACTGGT, ACTATCGAG

output : 6

Fundamentals of Algorithmic Problem Solving

Choosing an appropiate methods to specify the Algorithm

Fundamentals of Algorithmic Problem Solving

Choosing an appropiate methods to specify the Algorithm

- two common ways of specifying an algorithm: in words, in pseudocode

Fundamentals of Algorithmic Problem Solving

Choosing an appropiate methods to specify the Algorithm

- two common ways of specifying an algorithm: in words, in pseudocode
- design an algorithm to find the greatest common divisor of two nonnegative (not both of them zero) integers

Fundamentals of Algorithmic Problem Solving

Choosing an appropiate methods to specify the Algorithm

- two common ways of specifying an algorithm: in words, in pseudocode
- design an algorithm to find the greatest common divisor of two nonnegative (not both of them zero) integers
- **Euclid's Algorithm for the integers m, n**

Fundamentals of Algorithmic Problem Solving

Choosing an appropiate methods to specify the Algorithm

- two common ways of specifying an algorithm: in words, in pseudocode
- design an algorithm to find the greatest common divisor of two nonnegative (not both of them zero) integers
- **Euclid's Algorithm for the integers m, n**

Step 1 : if n = 0, return m; otherwise go to Step 2

Step 2 : divide m by n and set the variable r to the remainder

Step 3 : set m as n and n as r, go to Step 1

Fundamentals of Algorithmic Problem Solving

Choosing an appropiate methods to specify the Algorithm

- two common ways of specifying an algorithm: in words, in pseudocode
- design an algorithm to find the greatest common divisor of two nonnegative (not both of them zero) integers
- **Euclid's Algorithm for the integers m, n**

Step 1 : if n = 0, return m; otherwise go to Step 2

Step 2 : divide m by n and set the variable r to the remainder

Step 3 : set m as n and n as r, go to Step 1

- **Euclid(m,n)**

input : two non-negative, not-both-zero integers m and n
output: the greatest common divisor of m and n

while $\mathrm{n} \neq 0$

$r \leftarrow m \ (mod\ n)$

$\mathrm{m} \leftarrow n$

$\mathrm{n} \leftarrow r$

return m

Fundamentals of Algorithmic Problem Solving

Choosing an appropiate methods to specify the Algorithm

- two common ways of specifying an algorithm: in words, in pseudocode
- design an algorithm to find the greatest common divisor of two nonnegative (not both of them zero) integers
- **Euclid's Algorithm for the integers m, n**

Step 1 : if n = 0, return m; otherwise go to Step 2

Step 2 : divide m by n and set the variable r to the remainder

Step 3 : set m as n and n as r, go to Step 1

- **Euclid(m,n)**

input : two non-negati

output: the greatest c

while $n \neq 0$

$r \leftarrow m \ (mod \ n)$

$m \leftarrow n$

$n \leftarrow r$

return m

- high-level description of algorithms that combines a natural language and familiar structures from a programming language
- use '←' for the assigments and '//' for the comments

Fundamentals of Algorithmic Problem Solving

Proving the correctness of the Algorithm

Fundamentals of Algorithmic Problem Solving

Proving the correctness of the Algorithm

- prove that the algorithm always returns the desired output for every legitimate input in a finite amount of time in a formal way

Fundamentals of Algorithmic Problem Solving

Proving the correctness of the Algorithm

- prove that the algorithm always returns the desired output for every legitimate input in a finite amount of time in a formal way
- use mathematical proof techniques such as proof by contradiction, induction, etc.

Fundamentals of Algorithmic Problem Solving

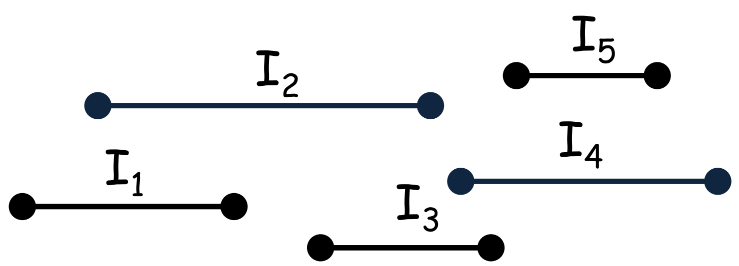
Proving the correctness of the Algorithm

- prove that the algorithm always returns the desired output for every legitimate input in a finite amount of time in a formal way
- use mathematical proof techniques such as proof by contradiction, induction, etc.
- suppose there are n meetings requests for a meeting room. Each meeting i has a starting time s_i and an ending time t_i. We have a constraint : no two meetings can be scheduled at same time. Design an algorithm that schedules as many meetings as possible to the room

Fundamentals of Algorithmic Problem Solving

Proving the correctness of the Algorithm

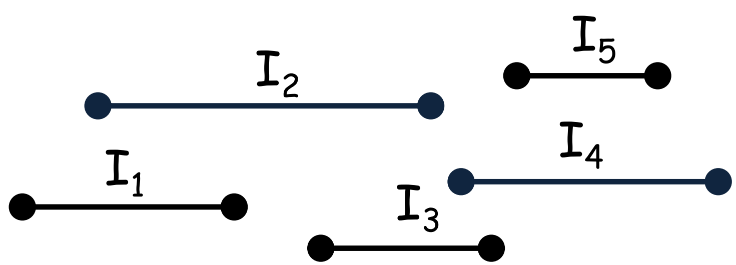
- prove that the algorithm always returns the desired output for every legitimate input in a finite amount of time in a formal way
- use mathematical proof techniques such as proof by contradiction, induction, etc.
- suppose there are n meetings requests for a meeting room. Each meeting i has a starting time s_i and an ending time t_i. We have a constraint : no two meetings can be scheduled at same time. Design an algorithm that schedules as many meetings as possible to the room



Fundamentals of Algorithmic Problem Solving

Proving the correctness of the Algorithm

- prove that the algorithm always returns the desired output for every legitimate input in a finite amount of time in a formal way
- use mathematical proof techniques such as proof by contradiction, induction, etc.
- suppose there are n meetings requests for a meeting room. Each meeting i has a starting time s_i and an ending time t_i. We have a constraint : no two meetings can be scheduled at same time. Design an algorithm that schedules as many meetings as possible to the room

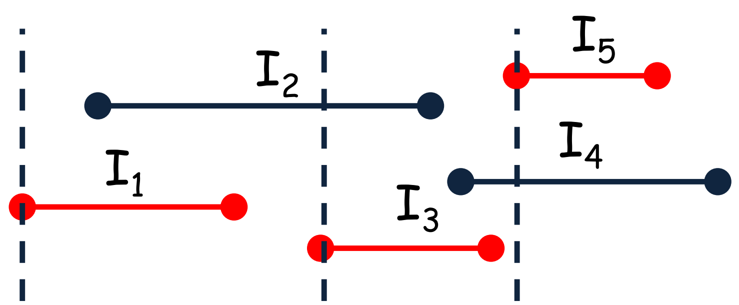


- choose the first interval as the one having the earliest start time
- remove all intervals not compatible with the chosen one

Fundamentals of Algorithmic Problem Solving

Proving the correctness of the Algorithm

- prove that the algorithm always returns the desired output for every legitimate input in a finite amount of time in a formal way
- use mathematical proof techniques such as proof by contradiction, induction, etc.
- suppose there are n meetings requests for a meeting room. Each meeting i has a starting time s_i and an ending time t_i. We have a constraint : no two meetings can be scheduled at same time. Design an algorithm that schedules as many meetings as possible to the room

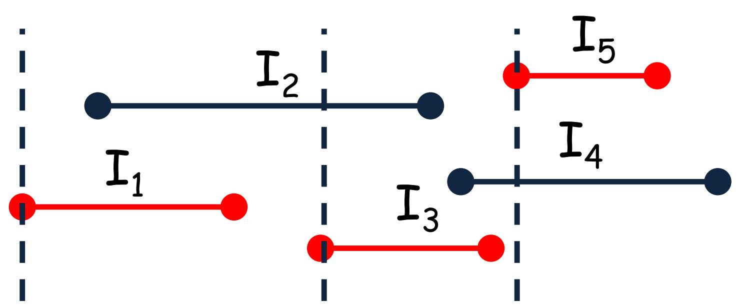


- choose the first interval as the one having the earliest start time
- remove all intervals not compatible with the chosen one

Fundamentals of Algorithmic Problem Solving

Proving the correctness of the Algorithm

- prove that the algorithm always returns the desired output for every legitimate input in a finite amount of time in a formal way
- use mathematical proof techniques such as proof by contradiction, induction, etc.
- suppose there are n meetings requests for a meeting room. Each meeting i has a starting time s_i and an ending time t_i. We have a constraint : no two meetings can be scheduled at same time. Design an algorithm that schedules as many meetings as possible to the room



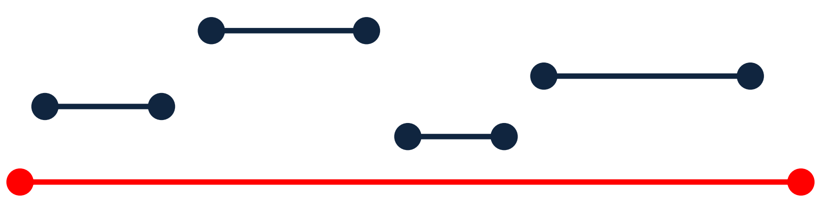
- choose the first interval as the one having the earliest start time
- remove all intervals not compatible with the chosen one

- one instance of inputs for which the algorithm works would not be enough to show the algorithm is correct

Fundamentals of Algorithmic Problem Solving

Proving the correctness of the Algorithm

- prove that the algorithm always returns the desired output for every legitimate input in a finite amount of time in a formal way
- use mathematical proof techniques such as proof by contradiction, induction, etc.
- suppose there are n meetings requests for a meeting room. Each meeting i has a starting time s_i and an ending time t_i. We have a constraint : no two meetings can be scheduled at same time. Design an algorithm that schedules as many meetings as possible to the room

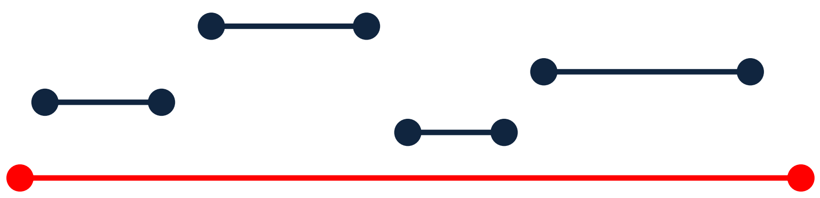


- choose the first interval as the one having the earliest start time
- remove all intervals not compatible with the chosen one

Fundamentals of Algorithmic Problem Solving

Proving the correctness of the Algorithm

- prove that the algorithm always returns the desired output for every legitimate input in a finite amount of time in a formal way
- use mathematical proof techniques such as proof by contradiction, induction, etc.
- suppose there are n meetings requests for a meeting room. Each meeting i has a starting time s_i and an ending time t_i. We have a constraint : no two meetings can be scheduled at same time. Design an algorithm that schedules as many meetings as possible to the room



- choose the first interval as the one having the earliest start time
- remove all intervals not compatible with the chosen one

- one instance of inputs for which the algorithm fails would be enough to show the algorithm is incorrect

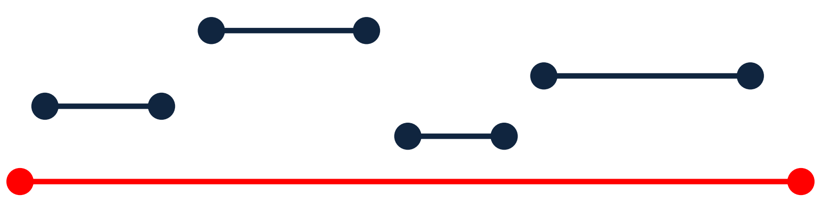
Fundamentals of Algorithmic Problem Solving

Proving the correctness of the Algorithm

- prove that the algorithm always returns the desired output for every legitimate input in a finite amount of time in a formal way

- use mathematical proof techniques such as proof by contradiction, induction, etc.

- suppose there are n meetings requests for a meeting room. Each meeting i has a starting time s_i and an ending time t_i. We have a constraint : no two meetings can be scheduled at same time. Design an algorithm that schedules as many meetings as possible to the room



- choose the first interval as the one having the earliest start time

- remove all intervals not compatible with the chosen one

- one instance of inputs for which the algorithm fails would be enough to show the algorithm is incorrect
  (however, failure to find such instance does not mean 'it is obvious' that the algorithm is correct)

Fundamentals of Algorithmic Problem Solving

Evaluating the efficiency of the Algorithm

Fundamentals of Algorithmic Problem Solving

Evaluating the efficiency of the Algorithm

- investigate algorithm's efficiency with respect to two resources: running time and memory space
  (time complexity and space complexity)

Fundamentals of Algorithmic Problem Solving

Evaluating the efficiency of the Algorithm

- investigate algorithm's efficiency with respect to two resources: running time and memory space
  (time complexity and space complexity)

- how long does the algorithm take to generate a desired output as a function of input size ?

Fundamentals of Algorithmic Problem Solving

Evaluating the efficiency of the Algorithm

- investigate algorithm's efficiency with respect to two resources: running time and memory space
  (time complexity and space complexity)

- how long does the algorithm take to generate a desired output as a function of input size ?

- how much working memory (typically RAM) required for the algorithm to terminate as a function of input size ?

Algorithm Efficiency

Units for Measuring Running Time

- some standard metrics such as second, millisecond can be used to measure the running time of an algorithm through the program implementing it

Algorithm Efficiency

Units for Measuring Running Time

- some standard metrics such as second, millisecond can be used to measure the running time of an algorithm through the program implementing it
  - the speed of a specific computer, or the quality of the program executing the algorithm, or the quality of the compiler may directly affect the running time

Algorithm Efficiency

Units for Measuring Running Time

- some standard metrics such as second, millisecond can be used to measure the running time of an algorithm through the program implementing it
  - the speed of a specific computer, or the quality of the program executing the algorithm, or the quality of the compiler may directly affect the running time
  - thus, the metric should be independent of these factors

Algorithm Efficiency

Units for Measuring Running Time

- some standard metrics such as second, millisecond can be used to measure the running time of an algorithm through the program implementing it
  - the speed of a specific computer, or the quality of the program executing the algorithm, or the quality of the compiler may directly affect the running time
  - thus, the metric should be independent of these factors
- identify the basic operations of the algorithm, and count the number of times the basic operations are executed on the input size

Algorithm Efficiency

Units for Measuring Running Time

- some standard metrics such as second, millisecond can be used to measure the running time of an algorithm through the program implementing it
  - the speed of a specific computer, or the quality of the program executing the algorithm, or the quality of the compiler may directly affect the running time
  - thus, the metric should be independent of these factors
- identify the basic operations of the algorithm, and count the number of times the basic operations are executed on the input size
- Let c_{op} be the execution time of an algorithm's basic operation and $C(n)$ be the number of times this operation is executed,

Algorithm Efficiency

Units for Measuring Running Time

- some standard metrics such as second, millisecond can be used to measure the running time of an algorithm through the program implementing it
  - the speed of a specific computer, or the quality of the program executing the algorithm, or the quality of the compiler may directly affect the running time
  - thus, the metric should be independent of these factors
- identify the basic operations of the algorithm, and count the number of times the basic operations are executed on the input size
- Let c_{op} be the execution time of an algorithm's basic operation and $C(n)$ be the number of times this operation is executed, the running time of the algorithm can be estimated by the formula

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

Algorithm Efficiency

Units for Measuring Running Time

- Let c_{op} be the execution time of an algorithm's basic operation and $C(n)$ be the number of times this operation is executed, the running time of the algorithm can be estimated by the formula

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

Algorithm Efficiency

Units for Measuring Running Time

- Let c_{op} be the execution time of an algorithm's basic operation and $C(n)$ be the number of times this operation is executed, the running time of the algorithm can be estimated by the formula

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

- How will the running time change if we double the input size ?

Algorithm Efficiency

Units for Measuring Running Time

- Let c_{op} be the execution time of an algorithm's basic operation and $C(n)$ be the number of times this operation is executed, the running time of the algorithm can be estimated by the formula

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

- How will the running time change if we double the input size ?
  - Let's observe how it changes depending on the input size

Algorithm Efficiency

Units for Measuring Running Time

- Let c_{op} be the execution time of an algorithm's basic operation and $C(n)$ be the number of times this operation is executed, the running time of the algorithm can be estimated by the formula

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

- How will the running time change if we double the input size ?
  - Let's observe how it changes depending on the input size
  - Assume $C(n) = \frac{1}{2}n^2$. Then $\frac{T(2n)}{T(n)} \approx \frac{C(2n)*c_{op}}{C(n)*c_{op}} \approx \frac{\frac{1}{2}(2n)^2}{\frac{1}{2}n^2} \approx 4$

Algorithm Efficiency

Units for Measuring Running Time

- Let c_{op} be the execution time of an algorithm's basic operation and $C(n)$ be the number of times this operation is executed, the running time of the algorithm can be estimated by the formula

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

- How will the running time change if we double the input size ?
  - Let's observe how it changes depending on the input size
  - Assume $C(n) = \frac{1}{2}n^2$. Then $\frac{T(2n)}{T(n)} \approx \frac{C(2n)*c_{op}}{C(n)*c_{op}} \approx \frac{\frac{1}{2}(2n)^2}{\frac{1}{2}n^2} \approx 4$
  - c_{op} and the multiplicative constant $\frac{1}{2}$ can be removed from the formula

    (we can answer such question without knowing them)

Algorithm Efficiency

Units for Measuring Running Time

- Let c_{op} be the execution time of an algorithm's basic operation and $C(n)$ be the number of times this operation is executed, the running time of the algorithm can be estimated by the formula

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

- How will the running time change if we double the input size ?
  - Let's observe how it changes depending on the input size
  - Assume $C(n) = \frac{1}{2}n^2$. Then $\frac{T(2n)}{T(n)} \approx \frac{C(2n)*c_{op}}{C(n)*c_{op}} \approx \frac{\frac{1}{2}(2n)^2}{\frac{1}{2}n^2} \approx 4$
  - c_{op} and the multiplicative constant $\frac{1}{2}$ can be removed from the formula

    (we can answer such question without knowing them)
  - focus on rate of growth of the function $T(n)$

Algorithm Efficiency

Order of Growth (Rate of Growth)

- For small inputs, a difference in running times can be ignored
  (it does not actually distinguish efficient algorithms from inefficient ones)
- For large inputs, a difference in running times becomes clear and remarkable

Algorithm Efficiency

Order of Growth (Rate of Growth)

- For small inputs, a difference in running times can be ignored (it does not actually distinguish efficient algorithms from inefficient ones)
- For large inputs, a difference in running times becomes clear and remarkable
- Growth rates of common functions measured in nanoseconds (assume each operation takes one nanosecond)

n $f(n)$	$\lg n$	n	$n \lg n$	n^2	2^n	$n!$
10	0.003 μs	0.01 μs	0.033 μs	0.1 μs	1 μs	3.63 ms
20	0.004 μs	0.02 μs	0.086 μs	0.4 μs	1 ms	77.1 years
30	0.005 μs	0.03 μs	0.147 μs	0.9 μs	1 sec	8.4×10^{15} yrs
40	0.005 μs	0.04 μs	0.213 μs	1.6 μs	18.3 min	
50	0.006 μs	0.05 μs	0.282 μs	2.5 μs	13 days	
100	0.007 μs	0.1 μs	0.644 μs	10 μs	4×10^{13} yrs	
1,000	0.010 μs	1.00 μs	9.966 μs	1 ms		
10,000	0.013 μs	10 μs	130 μs	100 ms		
100,000	0.017 μs	0.10 ms	1.67 ms	10 sec		
1,000,000	0.020 μs	1 ms	19.93 ms	16.7 min		
10,000,000	0.023 μs	0.01 sec	0.23 sec	1.16 days		
100,000,000	0.027 μs	0.10 sec	2.66 sec	115.7 days		
1,000,000,000	0.030 μs	1 sec	29.90 sec	31.7 years		

Algorithm Efficiency

Order of Growth (Rate of Growth)

- For small inputs, a difference in running times can be ignored
  (it does not actually distinguish efficient algorithms from inefficient ones)
- For large inputs, a difference in running times becomes clear and remarkable
- Growth rates of common functions measured in nanoseconds (assume each operation takes one nanosecond)

n $f(n)$	$\lg n$	n	$n \lg n$	n^2	2^n	$n!$
10	0.003 μs	0.01 μs	0.033 μs	0.1 μs	1 μs	3.63 ms
20	0.004 μs	0.02 μs	0.086 μs	0.4 μs	1 ms	77.1 years
30	0.005 μs	0.03 μs	0.147 μs	0.9 μs	1 sec	8.4×10^{15} yrs
40	0.005 μs	0.04 μs	0.213 μs	1.6 μs	18.3 min	
50	0.006 μs	0.05 μs	0.282 μs	2.5 μs	13 days	
100	0.007 μs	0.1 μs	0.644 μs	10 μs	4×10^{13} yrs	
1,000	0.010 μs	1.00 μs	9.966 μs	1 ms		
10,000	0.013 μs	10 μs	130 μs	100 ms		
100,000	0.017 μs	0.10 ms	1.67 ms	10 sec		
1,000,000	0.020 μs	1 ms	19.93 ms	16.7 min		
10,000,000	0.023 μs	0.01 sec	0.23 sec	1.16 days		
100,000,000	0.027 μs	0.10 sec	2.66 sec	115.7 days		
1,000,000,000	0.030 μs	1 sec	29.90 sec	31.7 years		

  - for $n = 10$, all such algorithms take roughly the same time

Algorithm Efficiency

Order of Growth (Rate of Growth)

- For small inputs, a difference in running times can be ignored (it does not actually distinguish efficient algorithms from inefficient ones)
- For large inputs, a difference in running times becomes clear and remarkable
- Growth rates of common functions measured in nanoseconds (assume each operation takes one nanosecond)

n $f(n)$	$\lg n$	n	$n \lg n$	n^2	2^n	$n!$
10	0.003 μs	0.01 μs	0.033 μs	0.1 μs	1 μs	3.63 ms
20	0.004 μs	0.02 μs	0.086 μs	0.4 μs	1 ms	77.1 years
30	0.005 μs	0.03 μs	0.147 μs	0.9 μs	1 sec	8.4×10^{15} yrs
40	0.005 μs	0.04 μs	0.213 μs	1.6 μs	18.3 min	
50	0.006 μs	0.05 μs	0.282 μs	2.5 μs	13 days	
100	0.007 μs	0.1 μs	0.644 μs	10 μs	4×10^{13} yrs	
1,000	0.010 μs	1.00 μs	9.966 μs	1 ms		
10,000	0.013 μs	10 μs	130 μs	100 ms		
100,000	0.017 μs	0.10 ms	1.67 ms	10 sec		
1,000,000	0.020 μs	1 ms	19.93 ms	16.7 min		
10,000,000	0.023 μs	0.01 sec	0.23 sec	1.16 days		
100,000,000	0.027 μs	0.10 sec	2.66 sec	115.7 days		
1,000,000,000	0.030 μs	1 sec	29.90 sec	31.7 years		

- for $n = 10$, all such algorithms take roughly the same time
- any algorithm with $n!$ running time becomes useless for $n \geq 20$

Algorithm Efficiency

Order of Growth (Rate of Growth)

- For small inputs, a difference in running times can be ignored (it does not actually distinguish efficient algorithms from inefficient ones)
- For large inputs, a difference in running times becomes clear and remarkable
- Growth rates of common functions measured in nanoseconds (assume each operation takes one nanosecond)

n $f(n)$	$\lg n$	n	$n \lg n$	n^2	2^n	$n!$
10	0.003 μs	0.01 μs	0.033 μs	0.1 μs	1 μs	3.63 ms
20	0.004 μs	0.02 μs	0.086 μs	0.4 μs	1 ms	77.1 years
30	0.005 μs	0.03 μs	0.147 μs	0.9 μs	1 sec	8.4×10^{15} yrs
40	0.005 μs	0.04 μs	0.213 μs	1.6 μs	18.3 min	
50	0.006 μs	0.05 μs	0.282 μs	2.5 μs	13 days	
100	0.007 μs	0.1 μs	0.644 μs	10 μs	4×10^{13} yrs	
1,000	0.010 μs	1.00 μs	9.966 μs	1 ms		
10,000	0.013 μs	10 μs	130 μs	100 ms		
100,000	0.017 μs	0.10 ms	1.67 ms	10 sec		
1,000,000	0.020 μs	1 ms	19.93 ms	16.7 min		
10,000,000	0.023 μs	0.01 sec	0.23 sec	1.16 days		
100,000,000	0.027 μs	0.10 sec	2.66 sec	115.7 days		
1,000,000,000	0.030 μs	1 sec	29.90 sec	31.7 years		

  - for $n = 10$, all such algorithms take roughly the same time
  - any algorithm with $n!$ running time becomes useless for $n \geq 20$
  - any algorithm with 2^n running time becomes impractical for $n > 40$

Algorithm Efficiency

Order of Growth (Rate of Growth)

- For small inputs, a difference in running times can be ignored
  (it does not actually distinguish efficient algorithms from inefficient ones)
- For large inputs, a difference in running times becomes clear and remarkable
- Growth rates of common functions measured in nanoseconds (assume each operation takes one nanosecond)

n $f(n)$	$\lg n$	n	$n \lg n$	n^2	2^n	$n!$
10	0.003 μs	0.01 μs	0.033 μs	0.1 μs	1 μs	3.63 ms
20	0.004 μs	0.02 μs	0.086 μs	0.4 μs	1 ms	77.1 years
30	0.005 μs	0.03 μs	0.147 μs	0.9 μs	1 sec	8.4×10^{15} yrs
40	0.005 μs	0.04 μs	0.213 μs	1.6 μs	18.3 min	
50	0.006 μs	0.05 μs	0.282 μs	2.5 μs	13 days	
100	0.007 μs	0.1 μs	0.644 μs	10 μs	4×10^{13} yrs	
1,000	0.010 μs	1.00 μs	9.966 μs	1 ms		
10,000	0.013 μs	10 μs	130 μs	100 ms		
100,000	0.017 μs	0.10 ms	1.67 ms	10 sec		
1,000,000	0.020 μs	1 ms	19.93 ms	16.7 min		
10,000,000	0.023 μs	0.01 sec	0.23 sec	1.16 days		
100,000,000	0.027 μs	0.10 sec	2.66 sec	115.7 days		
1,000,000,000	0.030 μs	1 sec	29.90 sec	31.7 years		

  - for $n = 10$, all such algorithms take roughly the same time
  - any algorithm with $n!$ running time becomes useless for $n \geq 20$
  - any algorithm with 2^n running time becomes impractical for $n > 40$
  - quadratic-time algorithms are practical up to $n = 1\, million$

Algorithm Efficiency

Order of Growth (Rate of Growth)

- For small inputs, a difference in running times can be ignored (it does not actually distinguish efficient algorithms from inefficient ones)
- For large inputs, a difference in running times becomes clear and remarkable
- Growth rates of common functions measured in nanoseconds (assume each operation takes one nanosecond)

n $f(n)$	$\lg n$	n	$n \lg n$	n^2	2^n	$n!$
10	0.003 μs	0.01 μs	0.033 μs	0.1 μs	1 μs	3.63 ms
20	0.004 μs	0.02 μs	0.086 μs	0.4 μs	1 ms	77.1 years
30	0.005 μs	0.03 μs	0.147 μs	0.9 μs	1 sec	8.4×10^{15} yrs
40	0.005 μs	0.04 μs	0.213 μs	1.6 μs	18.3 min	
50	0.006 μs	0.05 μs	0.282 μs	2.5 μs	13 days	
100	0.007 μs	0.1 μs	0.644 μs	10 μs	4×10^{13} yrs	
1,000	0.010 μs	1.00 μs	9.966 μs	1 ms		
10,000	0.013 μs	10 μs	130 μs	100 ms		
100,000	0.017 μs	0.10 ms	1.67 ms	10 sec		
1,000,000	0.020 μs	1 ms	19.93 ms	16.7 min		
10,000,000	0.023 μs	0.01 sec	0.23 sec	1.16 days		
100,000,000	0.027 μs	0.10 sec	2.66 sec	115.7 days		
1,000,000,000	0.030 μs	1 sec	29.90 sec	31.7 years		

  - for $n = 10$, all such algorithms take roughly the same time
  - any algorithm with $n!$ running time becomes useless for $n \geq 20$
  - any algorithm with 2^n running time becomes impractical for $n > 40$
  - quadratic-time algorithms are practical up to $n = 1\, million$
- by analyzing the order of growth of the function $T(n)$ that counts the algorithm's basic operation (simply considering the leading term of the function), we can evaluate whether a given algorithm is practical for a problem of a given size

Algorithm Efficiency

Worst-Case, Best-Case, Average-Case Analysis

Algorithm Efficiency

Worst-Case, Best-Case, Average-Case Analysis

Linear-Search(list,x)

input : $\{a_1, a_2, \ldots, a_n; x\}$
output: location

```
for i = 1 to n
    if x = a_i
    return i
return 0
```

Algorithm Efficiency

Worst-Case, Best-Case, Average-Case Analysis

Linear-Search(list,x)

input : $\{a_1, a_2, \ldots, a_n; x\}$
output: location

for i = 1 to n — n steps
    if $x = a_i$
    return i — 1 op
return 0

Algorithm Efficiency

Worst-Case, Best-Case, Average-Case Analysis

Linear-Search(list,x)

input : $\{a_1, a_2, \ldots, a_n; x\}$
output: location

for i = 1 to n — n steps
    if $x = a_i$
    return i
return 0

1 op

Worst Case :

- consider the worst-case input of size n for which the algorithm runs the longest among all possible inputs of same size

Algorithm Efficiency

Worst-Case, Best-Case, Average-Case Analysis

Linear-Search(list,x)

input : $\{a_1, a_2, \ldots, a_n; x\}$
output: location

for i = 1 to n — n steps
  if $x = a_i$
  return i
return 0

1 op

Worst Case :

- consider the worst-case input of size n for which the algorithm runs the longest among all possible inputs of same size

  (the element x matches the last one in the list, or the list does not contain the element x)

- $T(n) = n + 3$

Algorithm Efficiency

Worst-Case, Best-Case, Average-Case Analysis

Linear-Search(list,x)

input : $\{a_1, a_2, \ldots, a_n; x\}$
output: location

for i = 1 to n — n steps
    if $x = a_i$
    return i
return 0

1 op

Worst Case :

- consider the worst-case input of size n for which the algorithm runs the longest among all possible inputs of same size

  (the element x matches the last one in the list, or the list does not contain the element x)

- $T(n) = n + 3$

Best Case :

- consider the best-case input of size n for which the algorithm runs the fastest among all possible inputs of same size

Algorithm Efficiency

Worst-Case, Best-Case, Average-Case Analysis

Linear-Search(list,x)

input : $\{a_1, a_2, ..., a_n; x\}$
output: location

for i = 1 to n ——— n steps
    if $x = a_i$
    return i 1 op
return 0

Worst Case :

- consider the worst-case input of size n for which the algorithm runs the longest among all possible inputs of same size

  (the element x matches the last one in the list, or the list does not contain the element x)

- $T(n) = n + 3$

Best Case :

- consider the best-case input of size n for which the algorithm runs the fastest among all possible inputs of same size

  (the element x matches the first one in the list)

- $T(n) = 2$

Algorithm Efficiency

Worst-Case, Best-Case, Average-Case Analysis

Linear-Search(list,x)

input : $\{a_1, a_2, \ldots, a_n; x\}$
output: location

for i = 1 to n — n steps
    if $x = a_i$ — 1 op
    return i — 1 op
return 0 — 1 op

Average Case :

- neither the worst-case nor the best-case analysis gives us the necessary information about how the algorithm behaves on a random input
- it's the expected value for the number of operations

Worst Case :

- consider the worst-case input of size n for which the algorithm runs the longest among all possible inputs of same size

  (the element x matches the last one in the list, or the list does not contain the element x)

- $T(n) = n + 3$

Best Case :

- consider the best-case input of size n for which the algorithm runs the fastest among all possible inputs of same size

  (the element x matches the first one in the list)

- $T(n) = 2$

Algorithm Efficiency

Worst-Case, Best-Case, Average-Case Analysis

Linear-Search(list,x)

input : $\{a_1, a_2, \ldots, a_n; x\}$
output: location

for i = 1 to n — n steps
    if $x = a_i$
    return i — 1 op
return 0

- if $x = a_1$, then the algorithm terminates after 2 operations

  if $x = a_2$, then the algorithm terminates after 3 operations

  $\vdots$

  if $x = a_i$, then the algorithm terminates after $i + 1$ operation

  $\vdots$

  if $x = a_n$, then the algorithm terminates after $n + 1$ operations

  if $x \notin L$, then the algorithm terminates after $n + 1$ operations

Average Case :

Algorithm Efficiency

Worst-Case, Best-Case, Average-Case Analysis

Linear-Search(list,x)

input : $\{a_1, a_2, \ldots, a_n; x\}$
output: location

for i = 1 to n — n steps
  if $x = a_i$ — 1 op
  return i
return 0

- if $x = a_1$, then the algorithm terminates after 2 operations

  if $x = a_2$, then the algorithm terminates after 3 operations

  $\vdots$

  if $x = a_i$, then the algorithm terminates after $i + 1$ operation

  $\vdots$

  if $x = a_n$, then the algorithm terminates after $n + 1$ operations

  if $x \notin L$, then the algorithm terminates after $n + 1$ operations

Average Case :

- let p be the probability that $x \in L$, and $q = 1 - p$ be the probability that $x \notin L$

Algorithm Efficiency

Worst-Case, Best-Case, Average-Case Analysis

Linear-Search(list,x)

input : $\{a_1, a_2, \ldots, a_n; x\}$
output: location

for i = 1 to n — n steps
    if $x = a_i$ — 1 op
    return i
return 0

- if $x = a_1$, then the algorithm terminates after 2 operations

  if $x = a_2$, then the algorithm terminates after 3 operations

  $\vdots$

  if $x = a_i$, then the algorithm terminates after $i + 1$ operation

  $\vdots$

  if $x = a_n$, then the algorithm terminates after $n + 1$ operations

  if$x \notin L$, then the algorithm terminates after $n + 1$ operations

Average Case :

- let p be the probability that $x \in L$, and $q = 1 - p$ be the probability that $x \notin L$
- for each element a_i, the probability that $x = a_i$ is p/n

Algorithm Efficiency

Worst-Case, Best-Case, Average-Case Analysis

Linear-Search(list,x)

input : $\{a_1, a_2, \ldots, a_n; x\}$
output: location

for i = 1 to n — n steps
 if $x = a_i$
 return i
return 0

1 op

- if $x = a_1$, then the algorithm terminates after 2 operations

  if $x = a_2$, then the algorithm terminates after 3 operations

  $\vdots$

  if $x = a_i$, then the algorithm terminates after $i + 1$ operation

  $\vdots$

  if $x = a_n$, then the algorithm terminates after $n + 1$ operations

  if $x \notin L$, then the algorithm terminates after $n + 1$ operations

Average Case :

- let p be the probability that $x \in L$, and $q = 1 - p$ be the probability that $x \notin L$
- for each element a_i, the probability that $x = a_i$ is p/n
- the expected value for the number of operations

$$E(X) = \sum p(s).X(s)$$

Algorithm Efficiency

Worst-Case, Best-Case, Average-Case Analysis

Linear-Search(list,x)

input : $\{a_1, a_2, \ldots, a_n; x\}$
output: location

for i = 1 to n — n steps
    if $x = a_i$
    return i
return 0

1 op

- if $x = a_1$, then the algorithm terminates after 2 operations

  if $x = a_2$, then the algorithm terminates after 3 operations

  ⋮

  if $x = a_i$, then the algorithm terminates after $i + 1$ operation

  ⋮

  if $x = a_n$, then the algorithm terminates after $n + 1$ operations

  if$x \notin L$, then the algorithm terminates after $n + 1$ operations

Average Case :

- let p be the probability that $x \in L$, and $q = 1 - p$ be the probability that $x \notin L$
- for each element a_i, the probability that $x = a_i$ is p/n
- the expected value for the number of operations

$$E(X) = \sum p(s).X(s)$$
$$= 2.\frac{p}{n} + 3.\frac{p}{n} + \ldots + (n+1).\frac{p}{n} + (n+1).q = p\frac{(n+3)}{2} + q.(n+1)$$

Algorithm Efficiency

Worst-Case, Best-Case, Average-Case Analysis

Linear-Search(list,x)

input : $\{a_1, a_2, \ldots, a_n; x\}$
output: location

for i = 1 to n —— n steps
  if $x = a_i$ —— 1 op
  return i —— 1 op
return 0 —— 1 op

- if $x = a_1$, then the algorithm terminates after 2 operations

  if $x = a_2$, then the algorithm terminates after 3 operations

  ⋮

  if $x = a_i$, then the algorithm terminates after $i + 1$ operation

  ⋮

  if $x = a_n$, then the algorithm terminates after $n + 1$ operations

  if$x \notin L$, then the algorithm terminates after $n + 1$ operations

Average Case :

- let p be the probability that $x \in L$, and $q = 1 - p$ be the probability that $x \notin L$
- for each element a_i, the probability that $x = a_i$ is p/n
- the expected value for the number of operations

$$E(X) = \sum p(s).X(s)$$
$$= 2.\frac{p}{n} + 3.\frac{p}{n} + \ldots + (n+1).\frac{p}{n} + (n+1).q = p\frac{(n+3)}{2} + q.(n+1)$$

- for $p = 1$ and $q = 0$
  $E(X) = (n+3)/2$
- for $p = 0$ and $q = 1$
  $E(X) = n + 1$
- for $p = q = 1/2$
  $E(X) = (3n+5)/4$

Algorithm Efficiency

Worst-Case, Best-Case, Average-Case Analysis

Linear-Search(list,x)

input : $\{a_1, a_2, \ldots, a_n; x\}$
output: location

for i = 1 to n — n steps
  if $x = a_i$
  return i
return 0

- if $x = a_1$, then the algorithm terminates after 2 operations

  if $x = a_2$, then the algorithm terminates after 3 operations

  ⋮

  if $x = a_i$, then the algorithm terminates after $i + 1$ operation

  minates

  nates

- average-case analysis is more difficult than worst-case and best-case analysis
- applying the corresponding values to formula is easy, but probabilistic assumption for each particular case is hard to verify
- mostly deal with worst-case analysis

Average Case :

- let p be the pr
  $q = 1 - p$ be th
- for each element a_i, the probability that $x = a_i$ is p/n
- the expected value for the number of operations

$$E(X) = \sum p(s).X(s)$$
$$= 2.\frac{p}{n} + 3.\frac{p}{n} + \ldots + (n+1).\frac{p}{n} + (n+1).q = p\frac{(n+3)}{2} + q.(n+1)$$

$= 1$ and $q = 0$
$= (n+3)/2$

- for $p = 0$ and $q = 1$
  $E(X) = n + 1$
- for $p = q = 1/2$
  $E(X) = (3n+5)/4$

Asymptotic Notations

- for the algorithm's efficiency, we focus on the order of growth of the function that counts the algorithm's basic operations

Asymptotic Notations

- for the algorithm's efficiency, we focus on the order of growth of the function that counts the algorithm's basic operations

- to compare and rank such orders of growth, three common tools will be employed:

  - O (big-oh), asymptotic upper bound

  - Ω (big-omega), asymptotic lower bound

  - Θ (big-theta), asymptotic tight bound

Asymptotic Notations

- $O(g(n))$ is the class of all functions with a lower or same order of growth as $g(n)$

Asymptotic Notations

- O(g(n)) is the class of all functions with a lower or same order of growth as g(n)

$$24n + 21 \in O(n^2), 3n(n-1) \in O(n^2), 0.02n^3 + 0{,}04n^2 \notin O(n^2), n^4 \notin O(n^2)$$

Asymptotic Notations

- $O(g(n))$ is the class of all functions with a lower or same order of growth as $g(n)$

$$24n + 21 \in O(n^2),\ 3n(n-1) \in O(n^2),\ 0.02n^3 + 0{,}04n^2 \notin O(n^2),\ n^4 \notin O(n^2)$$

- $\Omega(g(n))$ is the class of all functions with a higher or same order of growth as $g(n)$

Asymptotic Notations

- O(g(n)) is the class of all functions with a lower or same order of growth as g(n)

$$24n + 21 \in O(n^2), 3n(n-1) \in O(n^2), 0.02n^3 + 0,04n^2 \notin O(n^2), n^4 \notin O(n^2)$$

- Ω(g(n)) is the class of all functions with a higher or same order of growth as g(n)

$$24n^3 \in \Omega(n^2), 3n(n-1) \in \Omega(n^2), 27n + 100 \notin \Omega(n^2)$$

Asymptotic Notations

- O(g(n)) is the class of all functions with a lower or same order of growth as g(n)

$$24n + 21 \in O(n^2), 3n(n-1) \in O(n^2), 0.02n^3 + 0{,}04n^2 \notin O(n^2), n^4 \notin O(n^2)$$

- Ω(g(n)) is the class of all functions with a higher or same order of growth as g(n)

$$24n^3 \in \Omega(n^2), 3n(n-1) \in \Omega(n^2), 27n + 100 \notin \Omega(n^2)$$

- Θ(g(n)) is the class of all functions with the same order of growth as g(n)

Asymptotic Notations

- O(g(n)) is the class of all functions with a lower or same order of growth as g(n)

$$24n + 21 \in O(n^2), 3n(n-1) \in O(n^2), 0.02n^3 + 0,04n^2 \notin O(n^2), n^4 \notin O(n^2)$$

- Ω(g(n)) is the class of all functions with a higher or same order of growth as g(n)

$$24n^3 \in \Omega(n^2), 3n(n-1) \in \Omega(n^2), 27n + 100 \notin \Omega(n^2)$$

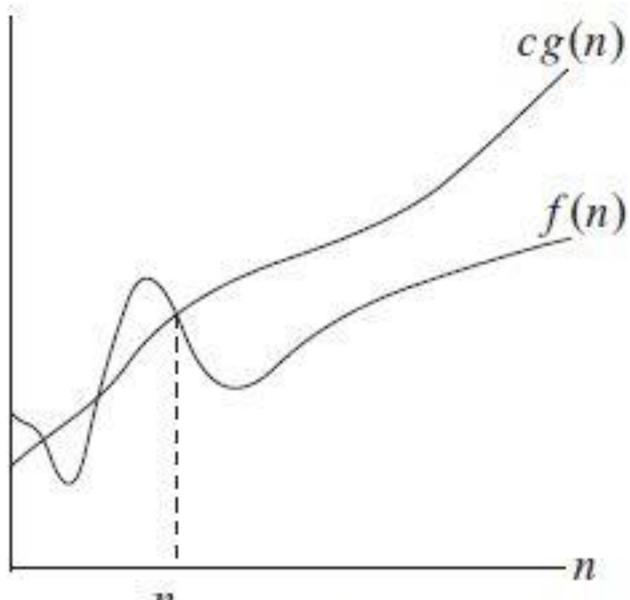
- Θ(g(n)) is the class of all functions with the same order of growth as g(n)

$$24n^2 + 17n \in \Theta(n^2), n^2 + 17\log n \in \Theta(n^2), 27n + 100 \notin \Theta(n^2), n^3 \notin \Theta(n^2)$$

Big-Oh Notation

Definition : Let $f, g : \mathbb{Z}^+ \to \mathbb{R}$ be two functions. If there are constants C and n_0 such that $|f(n)| \leq C.|g(n)|$ for all $n \in \mathbb{Z}$ where $n \geq n_0$, we say that f is big-oh of g,

$$f(n) = O(g(n))$$

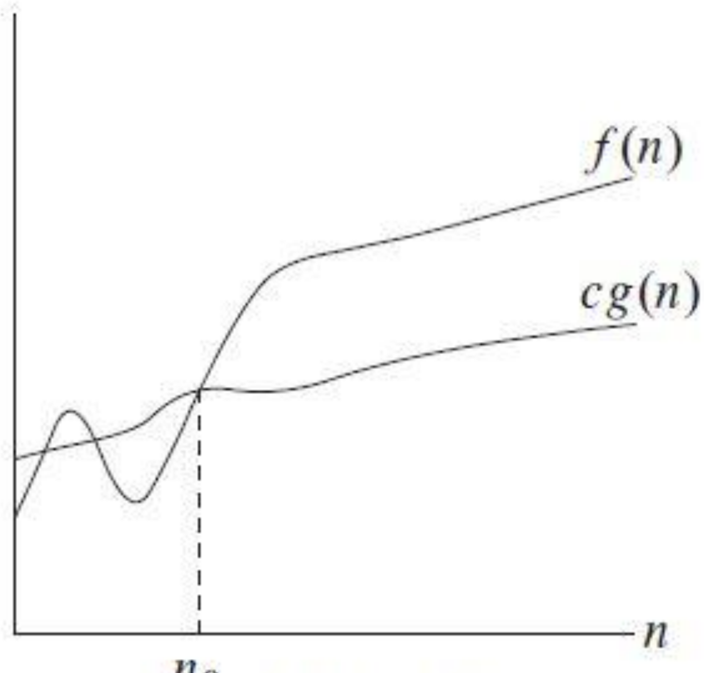


$f(n) = O(g(n))$

Big-Omega Notation

Definition : Let $f, g : \mathbb{Z}^{+} \rightarrow \mathbb{R}$ be two functions. If there are constants C and n_0 such that $|f(n)| \geq C.|g(n)|$ for all $n \in \mathbb{Z}$ where $n \geq n_0$, we say that f is big-omega of g,

$$f(n) = \Omega(g(n))$$

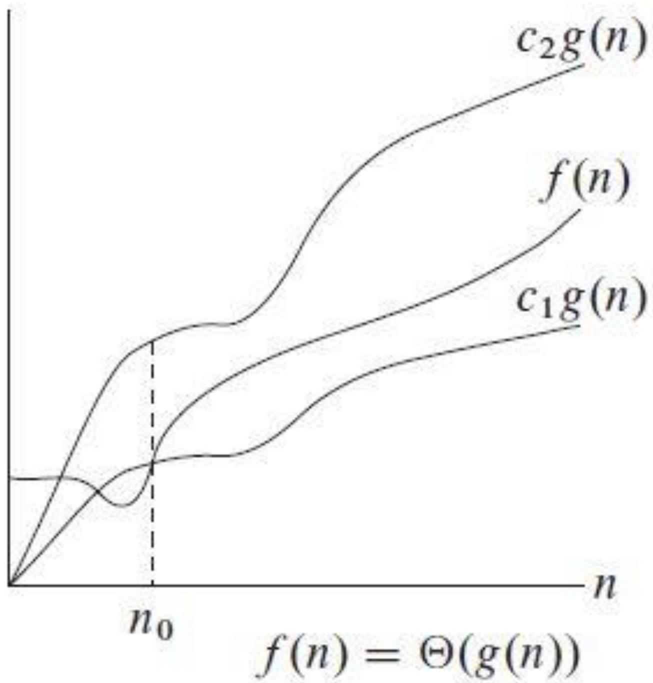


$f(n) = \Omega(g(n))$

Big-Theta Notation

Definition : Let $f, g : \mathbb{Z}^+ \to \mathbb{R}$ be two functions. If there are constants C_1, C_2, and n_0 such that $C_1.|g(n)| \leq |f(n)| \leq C_2.|g(n)|$ for all $n \in \mathbb{Z}$ where $n \geq n_0$, we say that f is big-theta of g,

$$f(n) = \Theta(g(n))$$



Big-Oh Notation

- $f, g : \mathbb{Z}^+ \to \mathbb{R}$, $f(n) = 5n$ and $g(n) = n^2$.

Big-Oh Notation

- $f, g : \mathbb{Z}^+ \rightarrow \mathbb{R}$, $f(n) = 5n$ and $g(n) = n^2$.
  - $f(1) = 5, f(2) = 10, f(3) = 15, f(4) = 20, f(5) = 25, \ldots$
    $g(1) = 1, g(2) = 4, g(3) = 9, g(4) = 16, g(5) = 25, \ldots$

Big-Oh Notation

- $f, g : \mathbb{Z}^+ \rightarrow \mathbb{R}$, $f(n) = 5n$ and $g(n) = n^2$.
  - $f(1) = 5, f(2) = 10, f(3) = 15, f(4) = 20, f(5) = 25, \ldots$
    $g(1) = 1, g(2) = 4, g(3) = 9, g(4) = 16, g(5) = 25, \ldots$
  - for $n \geq 5, n^2 \geq 5n \rightarrow |f(n)| \leq |g(n)|$

Big-Oh Notation

- $f, g : \mathbb{Z}^+ \to \mathbb{R}$, $f(n) = 5n$ and $g(n) = n^2$.
  - $f(1) = 5, f(2) = 10, f(3) = 15, f(4) = 20, f(5) = 25, \ldots$
    $g(1) = 1, g(2) = 4, \; g(3) = 9, \;\; g(4) = 16, g(5) = 25, \ldots$
  - for $n \geq 5$, $n^2 \geq 5n \to |f(n)| \leq |g(n)|$
  - for $C = 1$ and $n_0 = 5$,

    $|f(n)| \leq C.|g(n)|$ for all $n \geq n_0$. Thus, $f(n) = O(g(n))$.

Big-Oh Notation

- $f, g : \mathbb{Z}^+ \rightarrow \mathbb{R}$, $f(n) = 5n$ and $g(n) = n^2$.
  - $f(1) = 5, f(2) = 10, f(3) = 15, f(4) = 20, f(5) = 25, \ldots$
    $g(1) = 1, g(2) = 4,\ g(3) = 9,\ \ g(4) = 16, g(5) = 25, \ldots$
  - for $n \geq 5$, $n^2 \geq 5n \rightarrow |f(n)| \leq |g(n)|$
  - for $C = 1$ and $n_0 = 5$,

    $|f(n)| \leq C.|g(n)|$ for all $n \geq n_0$. Thus, $f(n) = O(g(n))$.
  - C and n_0 don't have to be unique

Big-Oh Notation

- $f, g : \mathbb{Z}^+ \rightarrow \mathbb{R}$, $f(n) = 5n^2 + 3n + 1$ and $g(n) = n^2$.

Big-Oh Notation

- $f, g : \mathbb{Z}^+ \to \mathbb{R}$, $f(n) = 5n^2 + 3n + 1$ and $g(n) = n^2$.

  $|f(n)| = |5n^2 + 3n + 1|$

Big-Oh Notation

- $f, g : \mathbb{Z}^+ \rightarrow \mathbb{R}$, $f(n) = 5n^2 + 3n + 1$ and $g(n) = n^2$.

  $|f(n)| = |5n^2 + 3n + 1| = 5n^2 + 3n + 1$

Big-Oh Notation

- $f, g : \mathbb{Z}^+ \to \mathbb{R}$, $f(n) = 5n^2 + 3n + 1$ and $g(n) = n^2$.

$$
\begin{aligned}
|f(n)| = |5n^2 + 3n + 1| &= 5n^2 + 3n + 1 \\
&\leq 5n^2 + 3n^2 + n^2
\end{aligned}
$$

Big-Oh Notation

- $f, g : \mathbb{Z}^+ \to \mathbb{R}$, $f(n) = 5n^2 + 3n + 1$ and $g(n) = n^2$.

$$|f(n)| = |5n^2 + 3n + 1| = 5n^2 + 3n + 1$$
$$\leq 5n^2 + 3n^2 + n^2 = 9n^2$$

Big-Oh Notation

- $f, g : \mathbb{Z}^+ \to \mathbb{R}$, $f(n) = 5n^2 + 3n + 1$ and $g(n) = n^2$.

$$|f(n)| = |5n^2 + 3n + 1| = 5n^2 + 3n + 1$$
$$\leq 5n^2 + 3n^2 + n^2 = 9n^2 = 9|g(n)|$$

Big-Oh Notation

- $f, g : \mathbb{Z}^+ \to \mathbb{R}$, $f(n) = 5n^2 + 3n + 1$ and $g(n) = n^2$.

  $$|f(n)| = |5n^2 + 3n + 1| = 5n^2 + 3n + 1$$
  $$\leq 5n^2 + 3n^2 + n^2 = 9n^2 = 9|g(n)|$$

  for $C = 9$ and $n_0 = 1$,

  $|f(n)| \leq C.|g(n)|$ for all $n \geq n_0$. Thus, $f(n) = O(g(n))$.

Big-Oh Notation

- $f, g : \mathbb{Z}^+ \to \mathbb{R}$, $f(n) = 5n^2 + 3n + 1$ and $g(n) = n^2$.

  $|f(n)| = |5n^2 + 3n + 1| = 5n^2 + 3n + 1$

  $\leq 5n^2 + 3n^2 + n^2 = 9n^2 = 9|g(n)|$

  for $C = 9$ and $n_0 = 1$,

  $|f(n)| \leq C.|g(n)|$ for all $n \geq n_0$. Thus, $f(n) = O(g(n))$.

  $|g(n)| = |n^2| = n^2$

Big-Oh Notation

- $f, g : \mathbb{Z}^+ \to \mathbb{R}$, $f(n) = 5n^2 + 3n + 1$ and $g(n) = n^2$.

$$|f(n)| = |5n^2 + 3n + 1| = 5n^2 + 3n + 1$$
$$\leq 5n^2 + 3n^2 + n^2 = 9n^2 = 9|g(n)|$$

for $C = 9$ and $n_0 = 1$,

$|f(n)| \leq C.|g(n)|$ for all $n \geq n_0$. Thus, $f(n) = O(g(n))$.

$$|g(n)| = |n^2| = n^2 \leq 5n^2$$

Big-Oh Notation

- $f, g : \mathbb{Z}^+ \to \mathbb{R}$, $f(n) = 5n^2 + 3n + 1$ and $g(n) = n^2$.

$$|f(n)| = |5n^2 + 3n + 1| = 5n^2 + 3n + 1$$
$$\leq 5n^2 + 3n^2 + n^2 = 9n^2 = 9|g(n)|$$

for $C = 9$ and $n_0 = 1$,

$|f(n)| \leq C.|g(n)|$ for all $n \geq n_0$. Thus, $f(n) = O(g(n))$.

$$|g(n)| = |n^2| = n^2 \leq 5n^2 \leq \ 5n^2 + 3n + 1 = |f(n)|$$

Big-Oh Notation

- $f, g : \mathbb{Z}^+ \to \mathbb{R}$, $f(n) = 5n^2 + 3n + 1$ and $g(n) = n^2$.

$$|f(n)| = |5n^2 + 3n + 1| = 5n^2 + 3n + 1$$

$$\leq 5n^2 + 3n^2 + n^2 = 9n^2 = 9|g(n)|$$

for $C = 9$ and $n_0 = 1$,

$|f(n)| \leq C.|g(n)|$ for all $n \geq n_0$. Thus, $f(n) = O(g(n))$.

$$|g(n)| = |n^2| = n^2 \leq 5n^2 \leq 5n^2 + 3n + 1 = |f(n)|$$

for $C = 1$ and $n_0 = 1$,

$|g(n)| \leq C.|f(n)|$ for all $n \geq n_0$. Thus, $g(n) = O(f(n))$.

Big-Oh Notation

- $f, g : \mathbb{Z}^+ \to \mathbb{R}$, $f(n) = 7n^2$ and $g(n) = n^3$.

Big-Oh Notation

- $f, g : \mathbb{Z}^+ \rightarrow \mathbb{R}$, $f(n) = 7n^2$ and $g(n) = n^3$.

  $|f(n)| = |7n^2| = 7n^2$

Big-Oh Notation

- $f, g : \mathbb{Z}^+ \rightarrow \mathbb{R}$, $f(n) = 7n^2$ and $g(n) = n^3$.

  $|f(n)| = |7n^2| = 7n^2 \leq 7n^3 = 7|g(n)|$

Big-Oh Notation

- $f, g : \mathbb{Z}^+ \to \mathbb{R}$, $f(n) = 7n^2$ and $g(n) = n^3$.

  $|f(n)| = |7n^2| = 7n^2 \leq 7n^3 = 7|g(n)|$

  for $C = 7$ and $n_0 = 1$,

  $|f(n)| \leq C.|g(n)|$ for all $n \geq n_0$. Thus, $f(n) = O(g(n))$.

Big-Oh Notation

- $f, g : \mathbb{Z}^{+} \to \mathbb{R}$, $f(n) = 7n^2$ and $g(n) = n^3$.

  $|f(n)| = |7n^2| = 7n^2 \leq 7n^3 = 7|g(n)|$

  for $C = 7$ and $n_0 = 1$,

  $|f(n)| \leq C.|g(n)|$ for all $n \geq n_0$. Thus, $f(n) = O(g(n))$.

  $|g(n)| = |n^3| = n^3 \leq C.7.n^2 = C.|f(n)|$

Big-Oh Notation

- $f, g : \mathbb{Z}^+ \to \mathbb{R}$, $f(n) = 7n^2$ and $g(n) = n^3$.

  $|f(n)| = |7n^2| = 7n^2 \leq 7n^3 = 7|g(n)|$

  for $C = 7$ and $n_0 = 1$,

  $|f(n)| \leq C.|g(n)|$ for all $n \geq n_0$. Thus, $f(n) = O(g(n))$.

  $|g(n)| = |n^3| = n^3 \leq C.7.n^2 = C.|f(n)| \to n \leq C.7$ for all $n \geq n_0$

Big-Oh Notation

- $f, g : \mathbb{Z}^+ \to \mathbb{R}$, $f(n) = 7n^2$ and $g(n) = n^3$.

  $|f(n)| = |7n^2| = 7n^2 \leq 7n^3 = 7|g(n)|$

  for $C = 7$ and $n_0 = 1$,

  $|f(n)| \leq C.|g(n)|$ for all $n \geq n_0$. Thus, $f(n) = O(g(n))$.

  $|g(n)| = |n^3| = n^3 \leq C.7.n^2 = C.|f(n)| \to n \leq C.7$ for all $n \geq n_0$

  there cannot be any C and n_0 that satisfy this inequality.

Big-Oh Notation

- $f : \mathbb{Z}^+ \rightarrow \mathbb{R}$, $f(n) = a_t n^t + a_{t-1} n^{t-1} + \ldots + a_1 n + a_0$

Big-Oh Notation

- $f : \mathbb{Z}^+ \rightarrow \mathbb{R}$, $f(n) = a_t n^t + a_{t-1} n^{t-1} + \ldots + a_1 n + a_0$

$|f(n)| = |a_t n^t + a_{t-1} n^{t-1} + \ldots + a_1 n + a_0| \leq |a_t n^t| + \ldots + |a_1 n| + |a_0|$

Big-Oh Notation

- $f : \mathbb{Z}^+ \to \mathbb{R},\ f(n) = a_t n^t + a_{t-1} n^{t-1} + \ldots + a_1 n + a_0$

$$|f(n)| = |a_t n^t + a_{t-1} n^{t-1} + \ldots + a_1 n + a_0| \leq |a_t n^t| + \ldots + |a_1 n| + |a_0|$$
$$= |a_t| . n^t + \ldots + |a_1| . n + |a_0|$$

Big-Oh Notation

- $f : \mathbb{Z}^+ \to \mathbb{R}$, $f(n) = a_t n^t + a_{t-1} n^{t-1} + \ldots + a_1 n + a_0$

$$
\begin{aligned}
|f(n)| = |a_t n^t + a_{t-1} n^{t-1} + \ldots + a_1 n + a_0| &\leq |a_t n^t| + \ldots + |a_1 n| + |a_0| \\
&= |a_t| . n^t + \ldots + |a_1| . n + |a_0| \\
&\leq |a_t| . n^t + \ldots + |a_1| . n^t + |a_0| . n^t
\end{aligned}
$$

Big-Oh Notation

- $f : \mathbb{Z}^+ \to \mathbb{R},\ f(n) = a_t n^t + a_{t-1} n^{t-1} + \ldots + a_1 n + a_0$

$$
\begin{aligned}
|f(n)| = |a_t n^t + a_{t-1} n^{t-1} + \ldots + a_1 n + a_0| &\leq |a_t n^t| + \ldots + |a_1 n| + |a_0| \\
&= |a_t|.\, n^t + \ldots + |a_1|.\, n + |a_0| \\
&\leq |a_t|.\, n^t + \ldots + |a_1|.\, n^t + |a_0|.\, n^t \\
&\leq (|a_t| + \ldots + |a_1| + |a_0|).\, n^t = C.\, |n^t|
\end{aligned}
$$

Big-Oh Notation

- $f: \mathbb{Z}^+ \to \mathbb{R}$, $f(n) = a_t n^t + a_{t-1} n^{t-1} + \ldots + a_1 n + a_0$

$$\begin{aligned} |f(n)| = |a_t n^t + a_{t-1} n^{t-1} + \ldots + a_1 n + a_0| &\leq |a_t n^t| + \ldots + |a_1 n| + |a_0| \\ &= |a_t|.n^t + \ldots + |a_1|.n + |a_0| \\ &\leq |a_t|.n^t + \ldots + |a_1|.n^t + |a_0|.n^t \\ &\leq (|a_t| + \ldots + |a_1| + |a_0|).n^t = C.|n^t| \end{aligned}$$

for $C = |a_t| + \ldots + |a_1| + |a_0|$ and $n_0 = 1$,
$|f(n)| \leq C.|n^t|$ for all $n \geq n_0$. Thus, $f(n) = O(n^t)$

Big-Oh Notation

- $f : \mathbb{Z}^+ \to \mathbb{R}$, $f(n) = 1 + 2 + \ldots + n$

  $|f(n)| = |1 + 2 + \ldots + \mathrm{n}\ | = 1 + 2 + \ldots + n \leq n + n + \ldots + n = |n^2|$
  for $C = 1$ and $n_0 = 1$,
  $|f(n)| \leq C.|n^2|$ for all $n \geq n_0$. Thus, $f(n) = O(n^2)$

- $f : \mathbb{Z}^+ \to \mathbb{R}$, $f(n) = 1^2 + 2^2 + \ldots + n^2$

  $|f(n)| = |1^2 + 2^2 + \ldots + n^2| = 1^2 + 2^2 + \ldots + n^2 \leq n^2 + n^2 + \ldots + n^2 = |n^3|$
  for $C = 1$ and $n_0 = 1$,
  $|f(n)| \leq C.|n^3|$ for all $n \geq n_0$. Thus, $f(n) = O(n^3)$

- $f : \mathbb{Z}^+ \to \mathbb{R}$, $f(x) = 1^t + 2^t + \ldots + n^t$

  $|f(n)| = |1^t + 2^t + \ldots + n^t| = 1^t + 2^t + \ldots + n^t \leq n^t + n^t + \ldots + n^t = |n^{t+1}|$
  for $C = 1$ and $n_0 = 1$,
  $|f(n)| \leq C.|n^{t+1}|$ for all $n \geq n_0$. Thus, $f(n) = O(n^{t+1})$

Big-Oh Notation

- Basic Efficieny Classes

1	: constant
$\log n$	: logarithmic
n	: linear
$n \log n$	: linearithmic (loglinear)
n^2	: quadratic
n^t	: polynomial
2^n	: exponential
$n!$	: factorial

Big-Oh Notation

- $f_1(n) = O(g_1(n))$ and $f_2(n) = O\big(g_2(n)\big)$

  $|f_1(n) + f_2(n)| \leq |f_1(n)| + |f_2(n)|$

Big-Oh Notation

- $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$

$$\begin{aligned} |f_1(n) + f_2(n)| &\leq |f_1(n)| + |f_2(n)| \\ &\leq C_1|g_1(n)| + C_2|g_2(n)| \end{aligned}$$

Big-Oh Notation

- $f_1(n) = O(g_1(n))$ and $f_2(n) = O\big(g_2(n)\big)$

$$
\begin{aligned}
|f_1(n) + f_2(n)| &\leq |f_1(n)| + |f_2(n)| \\
&\leq C_1|g_1(n)| + C_2|g_2(n)| \\
&\leq C_1|g(n)| + C_2|g(n)| \quad \text{where} \quad g(n) = max\ \{g_1(n), g_2(n)\}
\end{aligned}
$$

Big-Oh Notation

- $f_1(n) = O(g_1(n))$ and $f_2(n) = O\left(g_2(n)\right)$

$$
\begin{aligned}
|f_1(n) + f_2(n)| &\leq |f_1(n)| + |f_2(n)| \\
&\leq C_1|g_1(n)| + C_2|g_2(n)| \\
&\leq C_1|g(n)| + C_2|g(n)| \quad \text{where} \quad g(n) = max\ \{g_1(n), g_2(n)\} \\
&= (C_1+C_2)|g(n)|
\end{aligned}
$$

Big-Oh Notation

- $f_1(n) = O(g_1(n))$ and $f_2(n) = O\big(g_2(n)\big)$

$$
\begin{aligned}
|f_1(n) + f_2(n)| &\le |f_1(n)| + |f_2(n)| \\
&\le C_1|g_1(n)| + C_2|g_2(n)| \\
&\le C_1|g(n)| + C_2|g(n)| \quad \text{where} \quad g(n) = max\ \{g_1(n), g_2(n)\} \\
&= (C_1+C_2)|g(n)|
\end{aligned}
$$

$$f_1(n) + f_2(n) = O(max\ \{g_1(n), g_2(n)\})$$

Big-Oh Notation

- $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$

$$
\begin{aligned}
|f_1(n) + f_2(n)| &\le |f_1(n)| + |f_2(n)| \\
&\le C_1|g_1(n)| + C_2|g_2(n)| \\
&\le C_1|g(n)| + C_2|g(n)| \quad \text{where} \quad g(n) = max\ \{g_1(n), g_2(n)\} \\
&= (C_1 + C_2)|g(n)|
\end{aligned}
$$

$$f_1(n) + f_2(n) = O(max\ \{g_1(n), g_2(n)\})$$

$$f_1(n). f_2(n) = O(g_1(n).g_2(n))$$

Big-Oh Notation

- $f_1(n) = O(g_1(n))$ and $f_2(n) = O\big(g_2(n)\big)$

$$
\begin{aligned}
|f_1(n) + f_2(n)| &\le |f_1(n)| + |f_2(n)| \\
&\le C_1|g_1(n)| + C_2|g_2(n)| \\
&\le C_1|g(n)| + C_2|g(n)| \quad \text{where} \quad g(n) = max\ \{g_1(n), g_2(n)\} \\
&= (C_1+C_2)|g(n)|
\end{aligned}
$$

$$f_1(n) + f_2(n) = O(max\ \{g_1(n), g_2(n)\})$$

$$f_1(n). f_2(n) = O(g_1(n).g_2(n))$$

- $f(n) = (n+1)\log(n^2+1) + 3n^2$

Big-Oh Notation

- $f_1(n) = O(g_1(n))$ and $f_2(n) = O\left(g_2(n)\right)$

$$\begin{aligned} |f_1(n) + f_2(n)| &\leq |f_1(n)| + |f_2(n)| \\ &\leq C_1|g_1(n)| + C_2|g_2(n)| \\ &\leq C_1|g(n)| + C_2|g(n)| \quad \text{where} \quad g(n) = max\ \{g_1(n), g_2(n)\} \\ &= (C_1+C_2)|g(n)| \end{aligned}$$

$$f_1(n) + f_2(n) = O(max\ \{g_1(n), g_2(n)\})$$

$$f_1(n).f_2(n) = O(g_1(n).g_2(n))$$

- $f(n) = (n+1)\log(n^2+1) + 3n^2$

$O(n)$ $\qquad\qquad\qquad\qquad$ $O(n^2)$

Big-Oh Notation

- $f_1(n) = O(g_1(n))$ and $f_2(n) = O\big(g_2(n)\big)$

$$
\begin{aligned}
|f_1(n) + f_2(n)| &\le |f_1(n)| + |f_2(n)| \\
&\le C_1|g_1(n)| + C_2|g_2(n)| \\
&\le C_1|g(n)| + C_2|g(n)| \quad \text{where} \quad g(n) = max\ \{g_1(n), g_2(n)\} \\
&= (C_1+C_2)|g(n)|
\end{aligned}
$$

$$f_1(n) + f_2(n) = O(max\ \{g_1(n), g_2(n)\})$$

$$f_1(n).f_2(n) = O(g_1(n).g_2(n))$$

- $f(n) = (n+1)\log(n^2+1) + 3n^2$

$$\log(n^2+1) \le \log(2n^2)$$

$O(n)$ $\qquad$ $O(n^2)$

Big-Oh Notation

- $f_1(n) = O(g_1(n))$ and $f_2(n) = O\big(g_2(n)\big)$

$$
\begin{aligned}
|f_1(n) + f_2(n)| &\le |f_1(n)| + |f_2(n)| \\
&\le C_1|g_1(n)| + C_2|g_2(n)| \\
&\le C_1|g(n)| + C_2|g(n)| \quad \text{where} \quad g(n) = max\ \{g_1(n), g_2(n)\} \\
&= (C_1 + C_2)|g(n)|
\end{aligned}
$$

$$f_1(n) + f_2(n) = O(max\ \{g_1(n), g_2(n)\})$$

$$f_1(n). f_2(n) = O(g_1(n).g_2(n))$$

- $f(n) = (n + 1)\log(n^2 + 1) + 3n^2$

$O(n)$ $\quad$ $O(n^2)$

$$
\begin{aligned}
\log(n^2 + 1) &\le \log(2n^2) \\
&= \log 2 + \log n^2
\end{aligned}
$$

Big-Oh Notation

- $f_1(n) = O(g_1(n))$ and $f_2(n) = O\left(g_2(n)\right)$

$$
\begin{aligned}
|f_1(n) + f_2(n)| &\le |f_1(n)| + |f_2(n)| \\
&\le C_1|g_1(n)| + C_2|g_2(n)| \\
&\le C_1|g(n)| + C_2|g(n)| \quad \text{where} \quad g(n) = max\ \{g_1(n), g_2(n)\} \\
&= (C_1+C_2)|g(n)|
\end{aligned}
$$

$$f_1(n) + f_2(n) = O(max\ \{g_1(n), g_2(n)\})$$

$$f_1(n).f_2(n) = O(g_1(n).g_2(n))$$

- $f(n) = (n+1)\log(n^2+1) + 3n^2$

$O(n)$ $O(n^2)$

$$
\begin{aligned}
\log(n^2+1) &\le \log(2n^2) \\
&= \log 2 + \log n^2 \\
&= \log 2 + 2\log n
\end{aligned}
$$

Big-Oh Notation

- $f_1(n) = O(g_1(n))$ and $f_2(n) = O\big(g_2(n)\big)$

$$
\begin{aligned}
|f_1(n) + f_2(n)| &\le |f_1(n)| + |f_2(n)| \\
&\le C_1|g_1(n)| + C_2|g_2(n)| \\
&\le C_1|g(n)| + C_2|g(n)| \quad \text{where} \quad g(n) = max\,\{g_1(n), g_2(n)\} \\
&= (C_1 + C_2)|g(n)|
\end{aligned}
$$

$$f_1(n) + f_2(n) = O(max\,\{g_1(n), g_2(n)\})$$

$$f_1(n).\,f_2(n) = O(g_1(n).g_2(n))$$

- $f(n) = (n+1)\log(n^2+1) + 3n^2$

$O(n)$ $\quad O(\log n)$ $\quad O(n^2)$

$$
\begin{aligned}
\log(n^2+1) &\le \log(2n^2) \\
&= \log 2 + \log n^2 \\
&= \log 2 + 2\log n \\
&\le 3\log n
\end{aligned}
$$

Big-Oh Notation

- $f_1(n) = O(g_1(n))$ and $f_2(n) = O\left(g_2(n)\right)$

$$
\begin{aligned}
|f_1(n) + f_2(n)| &\leq |f_1(n)| + |f_2(n)| \\
&\leq C_1|g_1(n)| + C_2|g_2(n)| \\
&\leq C_1|g(n)| + C_2|g(n)| \quad \text{where} \quad g(n) = max\ \{g_1(n), g_2(n)\} \\
&= (C_1 + C_2)|g(n)|
\end{aligned}
$$

$$f_1(n) + f_2(n) = O(max\ \{g_1(n), g_2(n)\})$$

$$f_1(n). f_2(n) = O(g_1(n).g_2(n))$$

- $f(n) = (n+1)\log(n^2+1) + 3n^2$

$O(n)$ $\quad O(\log n)$ $\quad O(n^2)$

$$
\begin{aligned}
\log(n^2+1) &\leq \log(2n^2) \\
&= \log 2 + \log n^2 \\
&= \log 2 + 2\log n \\
&\leq 3\log n
\end{aligned}
$$

$$f(n) = O(n^2)$$

Worst-Case Analysis

Max-Integer(list)

input : $\{a_1, a_2, ..., a_n\}$
output: max of $\{a_1, a_2, ..., a_n\}$

```
max ← a_1
for i = 2 to n
    if max < a_i
        max ← a_i
return max
```

Worst-Case Analysis

Max-Integer(list)

input : $\{a_1, a_2, ..., a_n\}$
output: max of $\{a_1, a_2, ..., a_n\}$

max ← a_1
for i = 2 to n
    if max < a_i
        max ← a_i
return max

- T(n) : the number of operations the algorithm performs

Worst-Case Analysis

Max-Integer(list)

input : $\{a_1, a_2, ..., a_n\}$
output: max of $\{a_1, a_2, ..., a_n\}$

max ← a_1
for i = 2 to n
  if max < a_i
    max ← a_i
return max

2 op

- T(n) : the number of operations the algorithm performs
- the algorithm performs 2 operations on each execution of the loop

Worst-Case Analysis

Max-Integer(list)

input : $\{a_1, a_2, \ldots, a_n\}$
output: max of $\{a_1, a_2, \ldots, a_n\}$

max ← a_1
for i = 2 to n
    if max < a_i
        max ← a_i
return max

2 op

- T(n) : the number of operations the algorithm performs
- the algorithm performs 2 operations on each execution of the loop
- loop's variable increases from 2 to n (use sum formula)

Worst-Case Analysis

Max-Integer(list)

input : $\{a_1, a_2, ..., a_n\}$
output: max of $\{a_1, a_2, ..., a_n\}$

max ← a_1 — 1 op
for i = 2 to n
    if max < a_i
        max ← a_i — 2 op
return max

- T(n) : the number of operations the algorithm performs
- the algorithm performs 2 operations on each execution of the loop
- loop's variable increases from 2 to n (use sum formula)

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

Worst-Case Analysis

Max-Integer(list)

input : $\{a_1, a_2, ..., a_n\}$
output: max of $\{a_1, a_2, ..., a_n\}$

max ← a_1 1 op
for i = 2 to n
    if max < a_i
        max ← a_i 2 op
return max

- T(n) : the number of operations the algorithm performs
- the algorithm performs 2 operations on each execution of the loop
- loop's variable increases from 2 to n (use sum formula)

$$C(n) = \sum_{i=2}^{n} 2 + 1 = \sum_{i=1}^{n-1} 2 + 1$$

Worst-Case Analysis

Max-Integer(list)

input : $\{a_1, a_2, \ldots, a_n\}$
output: max of $\{a_1, a_2, \ldots, a_n\}$

max ← a_1 1 op
for i = 2 to n
  if max < a_i
    max ← a_i 2 op
return max

- T(n) : the number of operations the algorithm performs
- the algorithm performs 2 operations on each execution of the loop
- loop's variable increases from 2 to n (use sum formula)

$$C(n) = \sum_{i=2}^{n} 2 + 1 = \sum_{i=1}^{n-1} 2 + 1 = 2.(n-1) + 1 \in O(n)$$

Worst-Case Analysis

UniqueElements(list)

input : $\{a_1, a_2, \ldots, a_n\}$
output: return 'true' if all the elements are distinct; 'false' otherwise

```
for i = 1 to n – 1
    for j = i+1 to n
        if a_i = a_j
        return false
return true
```

Worst-Case Analysis

UniqueElements(list)

input : $\{a_1, a_2, ..., a_n\}$
output: return 'true' if all the elements are distinct; 'false' otherwise

for i = 1 to n – 1
    for j = i+1 to n
        if $a_i = a_j$
        return false   1 op
return true

- the algorithm performs 1 operation on each execution of the innermost loop

Worst-Case Analysis

UniqueElements(list)

input : $\{a_1, a_2, ..., a_n\}$
output: return 'true' if all the elements are distinct; 'false' otherwise

for i = 1 to n – 1
    for j = i+1 to n
        if $a_i = a_j$
        return false   1 op
return true

- the algorithm performs 1 operation on each execution of the innermost loop
- loop's variable increases from 1 to n – 1 for the outer loop, and from i + 1 to n for the innermost loop

Worst-Case Analysis

UniqueElements(list)

input : $\{a_1, a_2, \ldots, a_n\}$
output: return 'true' if all the elements are distinct; 'false' otherwise

for i = 1 to n – 1
  for j = i+1 to n
    if $a_i = a_j$
    return false   1 op
return true

- the algorithm performs 1 operation on each execution of the innermost loop
- loop's variable increases from 1 to n – 1 for the outer loop, and from i + 1 to n for the innermost loop

$$T(n) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} 1$$

Worst-Case Analysis

UniqueElements(list)

input : $\{a_1, a_2, \ldots, a_n\}$
output: return 'true' if all the elements are distinct; 'false' otherwise

for i = 1 to n – 1
  for j = i+1 to n
    if $a_i = a_j$
    return false   1 op
return true

- the algorithm performs 1 operation on each execution of the innermost loop
- loop's variable increases from 1 to n – 1 for the outer loop, and from i + 1 to n for the innermost loop

$$T(n) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} 1 = \sum_{i=1}^{n-1} [n - (i+1) + 1]$$

Worst-Case Analysis

UniqueElements(list)

input : $\{a_1, a_2, \ldots, a_n\}$
output: return 'true' if all the elements are distinct; 'false' otherwise

for i = 1 to n – 1
  for j = i+1 to n
    if $a_i = a_j$
    return false   1 op
return true

- the algorithm performs 1 operation on each execution of the innermost loop
- loop's variable increases from 1 to n – 1 for the outer loop, and from i + 1 to n for the innermost loop

$$T(n) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} 1 = \sum_{i=1}^{n-1} [n - (i+1) + 1] = \sum_{i=1}^{n-1} (n - i)$$

Worst-Case Analysis

UniqueElements(list)

input : $\{a_1, a_2, \ldots, a_n\}$
output: return 'true' if all the elements are distinct; 'false' otherwise

for i = 1 to n – 1
  for j = i+1 to n
    if $a_i = a_j$
    return false   1 op
return true

- the algorithm performs 1 operation on each execution of the innermost loop
- loop's variable increases from 1 to n – 1 for the outer loop, and from i + 1 to n for the innermost loop

$$T(n) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} 1 = \sum_{i=1}^{n-1} [n - (i+1) + 1] = \sum_{i=1}^{n-1} (n - i)$$

$$= \sum_{i=1}^{n-1} n - \sum_{i=1}^{n-1} i$$

Worst-Case Analysis

UniqueElements(list)

input : $\{a_1, a_2, \ldots, a_n\}$
output: return 'true' if all the elements are distinct; 'false' otherwise

for i = 1 to n – 1
    for j = i+1 to n
        if $a_i = a_j$
        return false   1 op
return true

- the algorithm performs 1 operation on each execution of the innermost loop
- loop's variable increases from 1 to n – 1 for the outer loop, and from i + 1 to n for the innermost loop

$$T(n) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} 1 = \sum_{i=1}^{n-1} [n - (i+1) + 1] = \sum_{i=1}^{n-1} (n - i)$$

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

Worst-Case Analysis

UniqueElements(list)

input : $\{a_1, a_2, ..., a_n\}$
output: return 'true' if all the elements are distinct; 'false' otherwise

for i = 1 to n – 1
    for j = i+1 to n
        if $a_i = a_j$
        return false   1 op
return true

- the algorithm performs 1 operation on each execution of the innermost loop
- loop's variable increases from 1 to n – 1 for the outer loop, and from i + 1 to n for the innermost loop

$$T(n) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} 1 = \sum_{i=1}^{n-1} [n - (i+1) + 1] = \sum_{i=1}^{n-1} (n-i)$$

$$= \sum_{i=1}^{n-1} n - \sum_{i=1}^{n-1} i = n(n-1) - \frac{n(n-1)}{2} = \frac{1}{2} n(n-1) \in O(n^2)$$

Worst-Case Analysis

mxn matrix A nxk matrix B mxk matrix C

$$\begin{pmatrix} a_{11} & \cdots & & a_{1n} \\ a_{i1} & a_{i2} & \cdots & a_{in} \\ a_{m1} & \cdots & & a_{mn} \end{pmatrix} . \begin{pmatrix} b_{11} & b_{1j} & b_{1k} \\ & b_{2j} & \\ \vdots & \vdots & \vdots \\ b_{n1} & b_{nj} & b_{nk} \end{pmatrix} = \begin{pmatrix} c_{11} & \cdots & c_{1k} \\ \vdots & \ddots & \vdots \\ c_{m1} & \cdots & c_{mk} \end{pmatrix}$$

$$c_{ij} = a_{i1}.b_{1j} + a_{i2}.b_{2j} + \cdots + a_{in}.b_{nj}$$

MatrixMultiplication(A,B)

input : two nxn matrices A,B
output: C = A.B

```
for i = 1 to n
    for j = 1 to n
        C[i,j] ← 0
        for k = 1 to n
            C[i,j] ← C[i,j] + A[i,k] * B[k,j]
return C
```

Worst-Case Analysis

MatrixMultiplication(A,B)

input : two nxn matrices A,B
output: A.B

```
for i = 1 to n
    for j = 1 to n
        C[i,j] ← 0
        for k = 1 to n
            C[i,j] ← C[i,j] + A[i,k] * B[k,j]
return C
```

Worst-Case Analysis

MatrixMultiplication(A,B)

input : two nxn matrices A,B
output: A.B

```
for i = 1 to n
    for j = 1 to n
        C[i,j] ← 0                                  1 op
        for k = 1 to n
            C[i,j] ← C[i,j] + A[i,k] * B[k,j]       3 op
return C
```

Worst-Case Analysis

MatrixMultiplication(A,B)

input : two nxn matrices A,B
output: A.B

for i = 1 to n
    for j = 1 to n
        $C[i,j] \leftarrow 0$ 1 op
        for k = 1 to n
            $C[i,j] \leftarrow C[i,j] + A[i,k] * B[k,j]$
return C

3 op

$$T(n) = \sum_{i=1}^{n} \sum_{j=1}^{n} (1 + \ \dots)$$

Worst-Case Analysis

MatrixMultiplication(A,B)

input : two nxn matrices A,B
output: A.B

for i = 1 to n
  for j = 1 to n
    $C[i,j] \leftarrow 0$ 1 op
    for k = 1 to n
      $C[i,j] \leftarrow C[i,j] + A[i,k] * B[k,j]$ 3 op
return C

$$T(n) = \sum_{i=1}^{n} \sum_{j=1}^{n} \left(1 + \sum_{k=1}^{n} 3\right)$$

Worst-Case Analysis

MatrixMultiplication(A,B)

input : two nxn matrices A,B
output: A.B

for i = 1 to n
    for j = 1 to n
        $C[i,j] \leftarrow 0$ 1 op
        for k = 1 to n
            $C[i,j] \leftarrow C[i,j] + A[i,k] * B[k,j]$
return C

3 op

$$T(n) = \sum_{i=1}^{n} \sum_{j=1}^{n} \left(1 + \sum_{k=1}^{n} 3\right) = \sum_{i=1}^{n} \sum_{j=1}^{n} (3n+1)$$

Worst-Case Analysis

MatrixMultiplication(A,B)

input : two nxn matrices A,B
output: A.B

for i = 1 to n
    for j = 1 to n
        $C[i,j] \leftarrow 0$ — 1 op
        for k = 1 to n
            $C[i,j] \leftarrow C[i,j] + A[i,k] * B[k,j]$ — 3 op
return C

$$T(n) = \sum_{i=1}^{n}\sum_{j=1}^{n}\left(1+\sum_{k=1}^{n}3\right) = \sum_{i=1}^{n}\sum_{j=1}^{n}(3n+1) = \sum_{i=1}^{n}n(3n+1)$$

Worst-Case Analysis

MatrixMultiplication(A,B)

input : two nxn matrices A,B
output: A.B

for i = 1 to n
    for j = 1 to n
        $C[i,j] \leftarrow 0$ 1 op
        for k = 1 to n
            $C[i,j] \leftarrow C[i,j] + A[i,k] * B[k,j]$ 3 op
return C

$$T(n) = \sum_{i=1}^{n} \sum_{j=1}^{n} \left(1 + \sum_{k=1}^{n} 3\right) = \sum_{i=1}^{n} \sum_{j=1}^{n} (3n+1) = \sum_{i=1}^{n} n(3n+1)$$
$$= n.n.(3n+1)$$

Worst-Case Analysis

MatrixMultiplication(A,B)

input : two nxn matrices A,B
output: A.B

for i = 1 to n
    for j = 1 to n
        $C[i,j] \leftarrow 0$ 1 op
        for k = 1 to n
            $C[i,j] \leftarrow C[i,j] + A[i,k] * B[k,j]$ 3 op
return C

$$T(n) = \sum_{i=1}^{n} \sum_{j=1}^{n} \left(1 + \sum_{k=1}^{n} 3\right) = \sum_{i=1}^{n} \sum_{j=1}^{n} (3n+1) = \sum_{i=1}^{n} n(3n+1)$$

$$= n.n.(3n+1) = 3n^3 + n^2 \in O(n^3)$$

Worst-Case Analysis

MatrixMultiplication(A,B)

input : two nxn matrices A,B
output: A.B

```
for i = 1 to n
    for j = 1 to n
        C[i,j] ← 0
        for k = 1 to n
            C[i,j] ← C[i,j] + A[i,k] * B[k,j]
return C
```

can be ignored

1 op

since constant number of operations can be counted as 1 operation, it can be ignored

3 op

$$T(n) = \sum_{i=1}^{n} \sum_{j=1}^{n} \left(1 + \sum_{k=1}^{n} 3\right) = \sum_{i=1}^{n} \sum_{j=1}^{n} (3n+1) = \sum_{i=1}^{n} n(3n+1)$$

$$= n.n.(3n+1) = 3n^3 + n^2 \in O(n^3)$$

$$T(n) = \sum_{i=1}^{n} \sum_{j=1}^{n} \sum_{k=1}^{n} 1 = n^3 \in O(n^3)$$