CSCI203 - Algorithms and Data Structures

Introduction to Algorithms

Japit Sionggo

sjapit@uow.edu.au

2 January 2023

Learning Outcome:

By the end of this lecture, you will be able to:

- Describe what an algorithm is;
- Explain what is an algorithm analysis;
- Perform algorithm analysis to determine the running-time complexity of an algorithm;
- Clarify some important mathematical functions commonly used in algorithm running-time complexity.

Algorithm and Data Structure

Introduction

Introduction

- One of the objective of this module is to learn about the analysis of algorithm. This analysis will enable you to select the best algorithm to perform a given task.
- Algorithms can be analysed in terms of the following aspects:
  - The correctness, that is, whether the algorithm performs a specification task according to a given specification.
  - The ease of understanding; in other words, the objective of what the algorithm is to perform is well defined. The input and output are well specified.

Introduction

- When executed, algorithms use processing and memory resources; they might also need bandwidth.
- Algorithms that perform their tasks correctly with the lowest resources consumption are naturally better candidates than those requiring more resources.

Introduction

- In this module, we will focus on the computational resource consumption of algorithms. In particular, we will focus on processing and memory resources.
- Processing requirements are usually measured in terms of the number of operations the central processing unit must carry out to execute the algorithm.
- The number of operations is important because the lower the number of operations the faster the algorithm is going to be executed.

Introduction

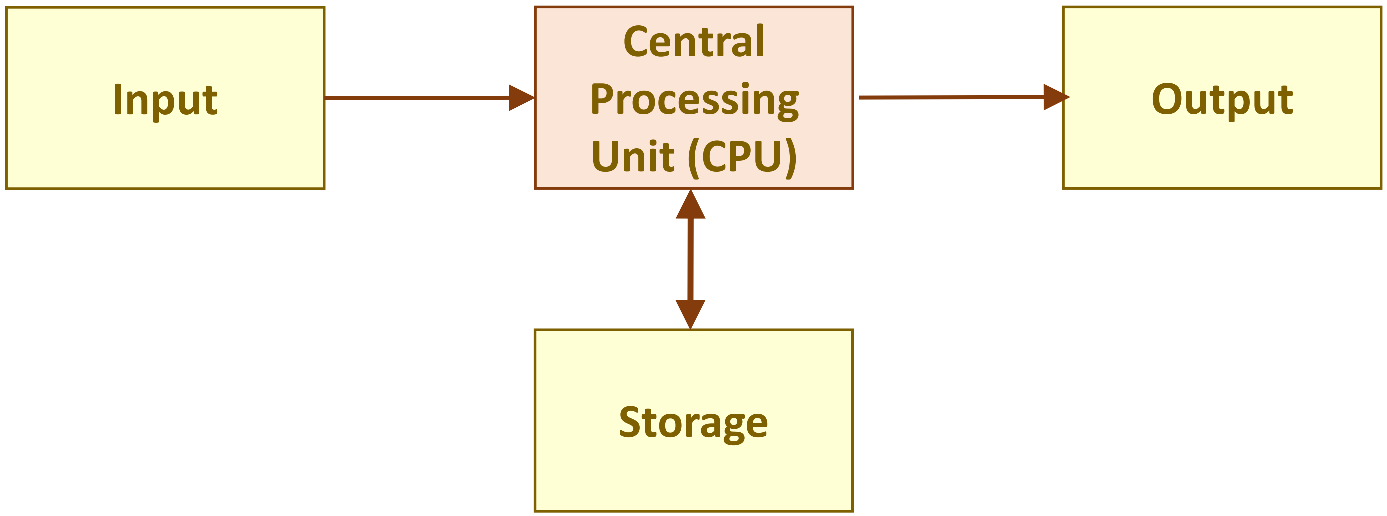
- Memory requirements are measured in terms of the number of memory positions required by the algorithms during its execution.
- Memory is also relevant because we have to make sure our algorithm requires an amount of memory that is well under the capacity of current computers.

A Simple RAM model

- Before we discuss how to perform the algorithm analysis, we will first look at a representation of the machine model on which we will be using it to execute our algorithm.
- Random Access Machine model, in short, the RAM model, is a simplified representation of reality of our modern day computer system. The model is simple enough to capture the aspects of the modern day computer that we need to analyse, but it is simple enough to allow us to do the analysis.

A Simple RAM model

- The model consists of an input unit, a central processing unit, an output unit, and a storage unit.



A Simple RAM model

- In this simplified RAM model, we have a central processing unit which can read and write data from the external environment. For example, the CPU can read data entered by a user using a keyboard, or it can display data on a screen. The CPU also has access to a storage unit, where it can write and read stored data.

Running time analysis assumptions

- Using this very simple model, we are going to estimate the running time of an algorithm according to the following four assumptions:
  - The machine has only one CPU, this means that instructions are executed sequentially
  - Each simple operation takes one unit of time. Simple operations include numerical operations such as addition, subtraction, multiplication, and division.

Running time analysis assumptions

- Other types of simple operations that take one time unit are
  - control instructions such as conditional branching, if else;
  - calling a function, (not executing the function itself, just calling it);
  - Writing or reading data from memory , and
  - Returning from a function.

Running time analysis assumptions

- Loop and functions are not considered as a simple operations. Hence, when we face a loop or a function in a code in our algorithm, we will have to go inside them and see how many simple operations they are made up of.
- Lastly, we assume that memory is unlimited, that is, we have as much memory as is needed in this model. (Note, in this simplified model, we do not consider memory hierarchy. In real computer, data that is stored in the cache memory is accessed much faster than data that is in the RAM memory. In turn, data in the RAM memory is accessed much faster than data stored in the hard disk.)

Memory consumption analysis assumptions

- When analysing memory consumption, we make only one assumption, that is, every simple variable, no matter its type, uses one memory position.

Getting Started

Anagram

Anagram

- An **anagram** is a word or phrase formed by rearranging the letters of a different word or phrase, typically using all the original letters exactly once.

The following are examples of anagram:

RAIL SAFETY	FAIRY TALES
CUSTOMERS	STORE SCUM
SILENT	LISTEN

http://www.enchantedlearning.com/english/anagram/numberofletters/5letters.shtml

Anagrams

Input: Two strings of characters

Output: True if the strings are anagrams on one another; False if they are not

We will consider two different algorithms to solve this problem

Algorithm 1

```
begin
    if string1 length != string2 length
        return False
    for all letters i in string1
        matched ← False
        for all letters j in string2
            if j is not marked and i = j
                mark j
                matched ← True
                Break
                #exit inner loop
        if matched = False
            return False
    return True
end
```

Algorithm 2

```
begin
    if string1 length != string2 length
        return False
    character count array ← 0
    for all letters i in string1
        increment the number of occurrences of i #increment:
        add 1 to
    for all letters j in string2
        decrement the number of occurrences of j #decrement:
        subtract 1 from
    for all integers k in the array
        if k != 0
            return False
    return True
end
```

Which one of these two algorithms is better?

Need to analyse the running-time efficiency of the two algorithms. (One of the learning outcomes of this subject.)

But how?

Algorithm analysis

- To **analyse an algorithm is to determine the amount of resources** (such as time and storage) necessary to execute it
  - (**Time** or **Space**) Complexity in terms of function $f(n)$. But why $f(n)$?
  - The running time of an algorithm typically grows with the input size. Hence the complexity of the algorithm is a function of n, that is $f(n)$.

Algorithm 1: (1/2)

```
begin
    if string1 length != string2 length
        return False
    for all letters i in string1
        matched ← False
        for all letters j in string2
            if j is not marked and i = j
                mark j
                matched ← True
                Break
                #exit inner loop
        if matched = False
            return False
    return True
end
```

Algorithm 1: (1/2)

```
begin
    if string1 length != string2 length
        return False
    for all letters i in string1
        matched ← False
        for all letters j in string2
            if j is not marked and i = j
                mark j
                matched ← True
                Break
                #exit inner loop
        if matched = False
            return False
    return True
end
```

1

Algorithm 1: (1/2)

```
begin
    if string1 length != string2 length
        return False
    for all letters i in string1
        matched ← False
        for all letters j in string2
            if j is not marked and i = j
                mark j
                matched ← True
                Break
                #exit inner loop
        if matched = False
            return False
    return True
end
```

1
a

Algorithm 1: (1/2)

```
begin
    if string1 length != string2 length
        return False
    for all letters i in string1
        matched ← False
        for all letters j in string2
            if j is not marked and i = j
                mark j
                matched ← True
                Break
                #exit inner loop
        if matched = False
            return False
    return True
end
```

1
a
$n+1$

Algorithm 1: (1/2)

```
begin
    if string1 length != string2 length
        return False
    for all letters i in string1
        matched ← False
        for all letters j in string2
            if j is not marked and i = j
                mark j
                matched ← True
                Break
                #exit inner loop
        if matched = False
            return False
    return True
end
```

1
a
$n+1$
n

Algorithm 1: (1/2)

```
begin
    if string1 length != string2 length
        return False
    for all letters i in string1
        matched ← False
        for all letters j in string2
            if j is not marked and i = j
                mark j
                matched ← True
                Break
                #exit inner loop
        if matched = False
            return False
    return True
end
```

1
a
$n+1$
n
$n \times (n+1)$

Algorithm 1: (1/2)

```
begin
    if string1 length != string2 length
        return False
    for all letters i in string1
        matched ← False
        for all letters j in string2
            if j is not marked and i = j
                mark j
                matched ← True
                Break
                #exit inner loop
        if matched = False
            return False
    return True
end
```

1
a
$n+1$
n
$n \times (n+1)$
$n \times n$

Algorithm 1: (1/2)

```
begin
    if string1 length != string2 length
        return False
    for all letters i in string1
        matched ← False
        for all letters j in string2
            if j is not marked and i = j
                mark j
                matched ← True
                Break
                #exit inner loop
        if matched = False
            return False
    return True
end
```

1
a
$n+1$
n
$n \times (n+1)$
$n \times n$
b

Algorithm 1: (1/2)

```
begin
    if string1 length != string2 length
        return False
    for all letters i in string1
        matched ← False
        for all letters j in string2
            if j is not marked and i = j
                mark j
                matched ← True
                Break
                #exit inner loop
        if matched = False
            return False
    return True
end
```

1
a
$n+1$
n
$n \times (n+1)$
$n \times n$
b
b

Algorithm 1: (1/2)

```
begin
    if string1 length != string2 length
        return False
    for all letters i in string1
        matched ← False
        for all letters j in string2
            if j is not marked and i = j
                mark j
                matched ← True
                Break
                #exit inner loop
        if matched = False
            return False
    return True
end
```

1
a
$n+1$
n
$n \times (n+1)$
$n \times n$
b
b
b

Algorithm 1: (1/2)

```
begin
    if string1 length != string2 length
        return False
    for all letters i in string1
        matched ← False
        for all letters j in string2
            if j is not marked and i = j
                mark j
                matched ← True
                Break
                #exit inner loop
        if matched = False
            return False
    return True
end
```

if string1 length != string2 length	1
return False	a
for all letters i in string1	$n+1$
matched ← False	n
for all letters j in string2	$n \times (n+1)$
if j is not marked and $i = j$	$n \times n$
mark j	b
matched ← True	b
Break	b
#exit inner loop	
if matched = False	n
return False	
return True	

Algorithm 1: (1/2)

begin	
if string1 length != string2 length	1
return False	a
for all letters i in string1	$n+1$
matched ← False	n
for all letters j in string2	$n \times (n+1)$
if j is not marked and $i = j$	$n \times n$
mark j	b
matched ← True	b
Break	b
#exit inner loop	
if matched = False	n
return False	c
return True	
end	

Algorithm 1: (1/2)

```
begin
    if string1 length != string2 length
        return False
    for all letters i in string1
        matched ← False
        for all letters j in string2
            if j is not marked and i = j
                mark j
                matched ← True
                Break
                #exit inner loop
        if matched = False
            return False
    return True
end
```

1
a
$n+1$
n
$n \times (n+1)$
$n \times n$
b
b
b
n
c
1

Algorithm 1: (2/2)

Sum up all those terms:

$T(n) = 1 + a + (n + 1) + n + (n^2 + n) + (n^2) + b + b + b + n + c + 1$

$T(n) = 3 + a + 3b + c + 4n + 2n^2$

$\boldsymbol{T(n) = 2n^2 + 4n + a + 3b + c + 3}$

Where:

a $= 0$ *or* 1 (worst case is 0)

b $= n$ (worst case is n), $\textit{that is, when the two words form an anagram.}$

c $= 0$ *or* 1 (worst case is 0)

For a worst-case scenario, we let $b = n$, hence, we can re-write the expression as $\boldsymbol{T(n) = 2n^2 + 7n + 3}$.

Algorithm 2:

```
begin
    if string1 length != string2 length
        return No
    character count array ← 0
    for all letters i in string1
        increment the number of occurrences of i
        #increment: add 1 to
    for all letters j in string2
        decrement the number of occurrences of j
        #decrement: subtract 1 from
    for all integers k in the array
        if k != 0
            return No
    return Yes
end
```

Algorithm 2: (1/13)

```
begin
    if string1 length != string2 length
        return No
    character count array ← 0
    for all letters i in string1
        increment the number of occurrences of i
        #increment: add 1 to
    for all letters j in string2
        decrement the number of occurrences of j
        #decrement: subtract 1 from
    for all integers k in the array
        if k != 0
            return No
    return Yes
end
```


Algorithm 2: (2/13)

```
begin
    if string1 length != string2 length
        return No
    character count array ← 0
    for all letters i in string1
        increment the number of occurrences of i
        #increment: add 1 to
    for all letters j in string2
        decrement the number of occurrences of j
        #decrement: subtract 1 from
    for all integers k in the array
        if k != 0
            return No
    return Yes
end
```

1

Algorithm 2: (3/13)

```
begin
    if string1 length != string2 length
        return No
    character count array ← 0
    for all letters i in string1
        increment the number of occurrences of i
        #increment: add 1 to
    for all letters j in string2
        decrement the number of occurrences of j
        #decrement: subtract 1 from
    for all integers k in the array
        if k != 0
            return No
    return Yes
end
```

1
a

Algorithm 2: (4/13)

```
begin
    if string1 length != string2 length
        return No
    character count array ← 0
    for all letters i in string1
        increment the number of occurrences of i
        #increment: add 1 to
    for all letters j in string2
        decrement the number of occurrences of j
        #decrement: subtract 1 from
    for all integers k in the array
        if k != 0
            return No
    return Yes
end
```

1
$\boldsymbol{a}$
1

Algorithm 2: (5/13)

```
begin
    if string1 length != string2 length
        return No
    character count array ← 0
    for all letters i in string1
        increment the number of occurrences of i
        #increment: add 1 to
    for all letters j in string2
        decrement the number of occurrences of j
        #decrement: subtract 1 from
    for all integers k in the array
        if k != 0
            return No
    return Yes
end
```

1
a
1
$n+1$

Algorithm 2: (6/13)

```
begin
    if string1 length != string2 length
        return No
    character count array ← 0
    for all letters i in string1
        increment the number of occurrences of i
        #increment: add 1 to
    for all letters j in string2
        decrement the number of occurrences of j
        #decrement: subtract 1 from
    for all integers k in the array
        if k != 0
            return No
    return Yes
end
```

1
a
1
$n+1$
n

Algorithm 2: (7/13)

```
begin
   if string1 length != string2 length
      return No
   character count array ← 0
   for all letters i in string1
      increment the number of occurrences of i
      #increment: add 1 to
   for all letters j in string2
      decrement the number of occurrences of j
      #decrement: subtract 1 from
   for all integers k in the array
      if k != 0
         return No
   return Yes
end
```

1
a
1
$n+1$
n
$n+1$

Algorithm 2: (8/13)

```
begin
    if string1 length != string2 length
        return No
    character count array ← 0
    for all letters i in string1
        increment the number of occurrences of i
        #increment: add 1 to
    for all letters j in string2
        decrement the number of occurrences of j
        #decrement: subtract 1 from
    for all integers k in the array
        if k != 0
            return No
    return Yes
end
```

1
a
1
$n+1$
n
$n+1$
n

Algorithm 2: (9/13)

```
begin
    if string1 length != string2 length
        return No
    character count array ← 0
    for all letters i in string1
        increment the number of occurrences of i
        #increment: add 1 to
    for all letters j in string2
        decrement the number of occurrences of j
        #decrement: subtract 1 from
    for all integers k in the array
        if k != 0
            return No
    return Yes
end
```

if string1 length != string2 length	1
return No	a
character count array ← 0	1
for all letters i in string1	$n+1$
increment the number of occurrences of i	n
#increment: add 1 to	
for all letters j in string2	$n+1$
decrement the number of occurrences of j	n
#decrement: subtract 1 from	
for all integers k in the array	27
if k != 0	
return No	
return Yes	

Algorithm 2: (10/13)

```
begin
    if string1 length != string2 length
        return No
    character count array ← 0
    for all letters i in string1
        increment the number of occurrences of i
        #increment: add 1 to
    for all letters j in string2
        decrement the number of occurrences of j
        #decrement: subtract 1 from
    for all integers k in the array
        if k != 0
            return No
    return Yes
end
```

if string1 length != string2 length	1
return No	a
character count array ← 0	1
for all letters i in string1	$n+1$
increment the number of occurrences of i	n
#increment: add 1 to	
for all letters j in string2	$n+1$
decrement the number of occurrences of j	n
#decrement: subtract 1 from	
for all integers k in the array	27
if k != 0	26
return No	
return Yes	

Algorithm 2: (11/13)

```
begin
    if string1 length != string2 length
        return No
    character count array ← 0
    for all letters i in string1
        increment the number of occurrences of i
        #increment: add 1 to
    for all letters j in string2
        decrement the number of occurrences of j
        #decrement: subtract 1 from
    for all integers k in the array
        if k != 0
            return No
    return Yes
end
```

1
a
1
$n+1$
n
$n+1$
n
27
26
b

Algorithm 2: (12/13)

```
begin
    if string1 length != string2 length
        return No
    character count array ← 0
    for all letters i in string1
        increment the number of occurrences of i
        #increment: add 1 to
    for all letters j in string2
        decrement the number of occurrences of j
        #decrement: subtract 1 from
    for all integers k in the array
        if k != 0
            return No
    return Yes
end
```

1
a
1
$n+1$
n
$n+1$
n
27
26
b
1

Algorithm 2: (13/13)

Sum up all those terms:

$$T(n) = 1 + a + 1 + (n + 1) + n + (n + 1) + n + 27 + 26 + b + 1$$

$$T(n) = 4n + 58 + a + b$$

$T(n) = 4n + 58$ (Worst case scenario)

Where:

$a = 0\ or\ 1$ (worst case is 0)

$b = 0\ or\ 1$ (worst case is 0)

Now what?

- From the analysis, we have:

Algorithm 1:

$$T(n) = 2n^2 + 7n + 3$$

Algorithm 2:

$$T(n) = 4n + 58$$

Which algorithm has a better running-time efficiency?

Run-time analysis - Comparison

- If the length of the strings are small, e.g., 3
- Algorithm 1:

$$T(n) = 2n^2 + 7n + 3$$

$$T(3) = 2(3)^2 + 7(3) + 3$$

$$= 18 + 21 + 3 = \mathbf{42} \text{ operation steps}$$

Run-time analysis - Comparison

- Algorithm 2:

  $T(n) = 4n + 58$

  $T(3) = 4(3) + 58 = \mathbf{70}$ operation steps

Run-time analysis - Comparison

- Hence, we may conclude that Algorithm 1 is better than Algorithm 2 because Algorithm 1 takes 42 steps (operations) to complete the algorithm while Algorithm 2 takes 70 steps to do the same.

Run-time analysis - Comparison

- Hence, we may conclude that Algorithm 1 is better than Algorithm 2 because Algorithm 1 takes 42 steps (operations) to complete the algorithm while Algorithm 2 takes 70 steps to do the same.
- But wait… is there a different, if the length of the string is larger? That is, what if the anagrams consists of, let's say 10-character words, or $n = 10$.

Run-time analysis - Comparison

- Hence, we may conclude that Algorithm 1 is better than Algorithm 2 because Algorithm 1 takes 42 steps (operations) to complete the algorithm while Algorithm 2 takes 70 steps to do the same.
- But wait… is there a different, if the length of the string is larger? That is, what if the anagrams consists of, let's say 10-character words, or $n = 10$.

The answer is **may not be!**
We will see with the next example.

Run-time analysis - Comparison

- If the length of the strings is bigger, e.g., 10
- Algorithm 1:

$$T(n) = 2n^2 + 7n + 3$$

$$T(10) = 2(10)^2 + 7(10) + 3$$

$$= 200 + 70 + 3 = \mathbf{273}$$ operation steps

- Algorithm 2:

$$T(n) = 4n + 58$$

$$T(10) = 4(10) + 58 = \mathbf{98}$$ operation steps

Run-time analysis - Comparison

- Hence, we conclude that Algorithm 2 is better than Algorithm 1 because Algorithm 2 takes 98 steps (operations) to complete the execution of the algorithm while Algorithm 1 takes 273 steps to do the same.

Huh… Why there is a contradicting conclusion?

Run-time analysis - Comparison

- Different algorithms can have dramatically different performance
- The performance difference often depends greatly on the size of the data set
- Choosing the right algorithm for the job at hand is, hence, important!

Run-time analysis - Comparison

- Different algorithms can have dramatically different performance.
- The performance difference often depends greatly on a couple of factors:
  - The size of the data set
  - The structure of the algorithm, e.g., the number of loop operations.

Run-time analysis - Comparison

- Choosing the right algorithm for the job at hand is, hence, important!

Run-time analysis - Comparison

- Choosing the right algorithm for the job at hand is, hence, important!

But if different data set and different algorithm structures produces different result or conclusion, how then can we know which algorithm is better (or algorithm performs better)?

Run-time analysis - Comparison

- In comparing algorithms, in general, we consider the *asymptotic behaviour* of the two algorithms for large problem sizes, under worst-case. (We will discuss asymptotic behaviour and worst-case in a while.)
- Uses a *high-level description of the algorithm* (Pseudo-code) to estimate the running-time efficiency as a function of input size n.

Run-time analysis - Comparison

- We have done an analysis. Had we, however, to undertake all the counting every time to analyse an algorithm, the task would be tedious and quickly become infeasible. We need some easier approach.

Complexity Analysis

Asymptotic behaviour

Run-time analysis - Comparison

- In comparing algorithms, in general, we consider the *asymptotic behaviour* of the two algorithms for large problem sizes, under worst-case. (We will discuss asymptotic behaviour and worst-case in a while.)
- Uses a *high-level description of the algorithm* (Pseudo-code) to estimate the running-time efficiency as a function of input size n.

Run-time analysis - Comparison

- Emphasize on the operation count's order of growth for large input sizes. (Note: the difference in running times on small inputs cannot really distinguish efficient algorithms from inefficient ones.)
- We want to know how the rate of growth in the number of operations performed by an algorithm as n grows, that is, the problem size grows.

Run-time analysis - Comparison

What is asymptotic time complexity?

The **limiting behaviour** of the execution time of an algorithm when the **size of the problem goes to very big**.

- We typically ignore small values of n, since we are usually interested in estimating how slow the program (algorithm) will be on large inputs.

Run-time analysis - Comparison

- A good rule of thumb is "the slower the asymptotic **growth rate**, the better the algorithm."
- **Growth rate** refers to how much more number of steps (operations) are required for an algorithm to complete its task as the number of data n gets bigger or increases.

Run-time analysis - Comparison

- As seen from our earlier calculation on the number of steps performed by Algorithm 1 and Algorithm 2:

N	Algorithm 1	Algorithm 2
3	42	70
10	273	98

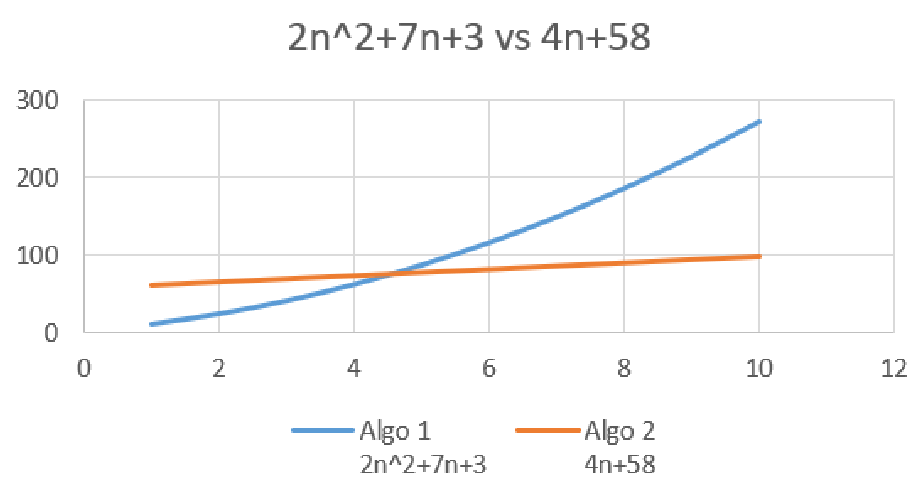
From the above-tabulated table, it is shown that the growth rate of Algorithm 1 is higher (or faster) than Algorithm 2. For an additional of 7 data, Algorithm 1 performs 231 additional operational steps while Algorithm 2 performs 28 additional operational steps.

The following table shown the various growth rate behaviour of function $f(n)$ as n get bigger

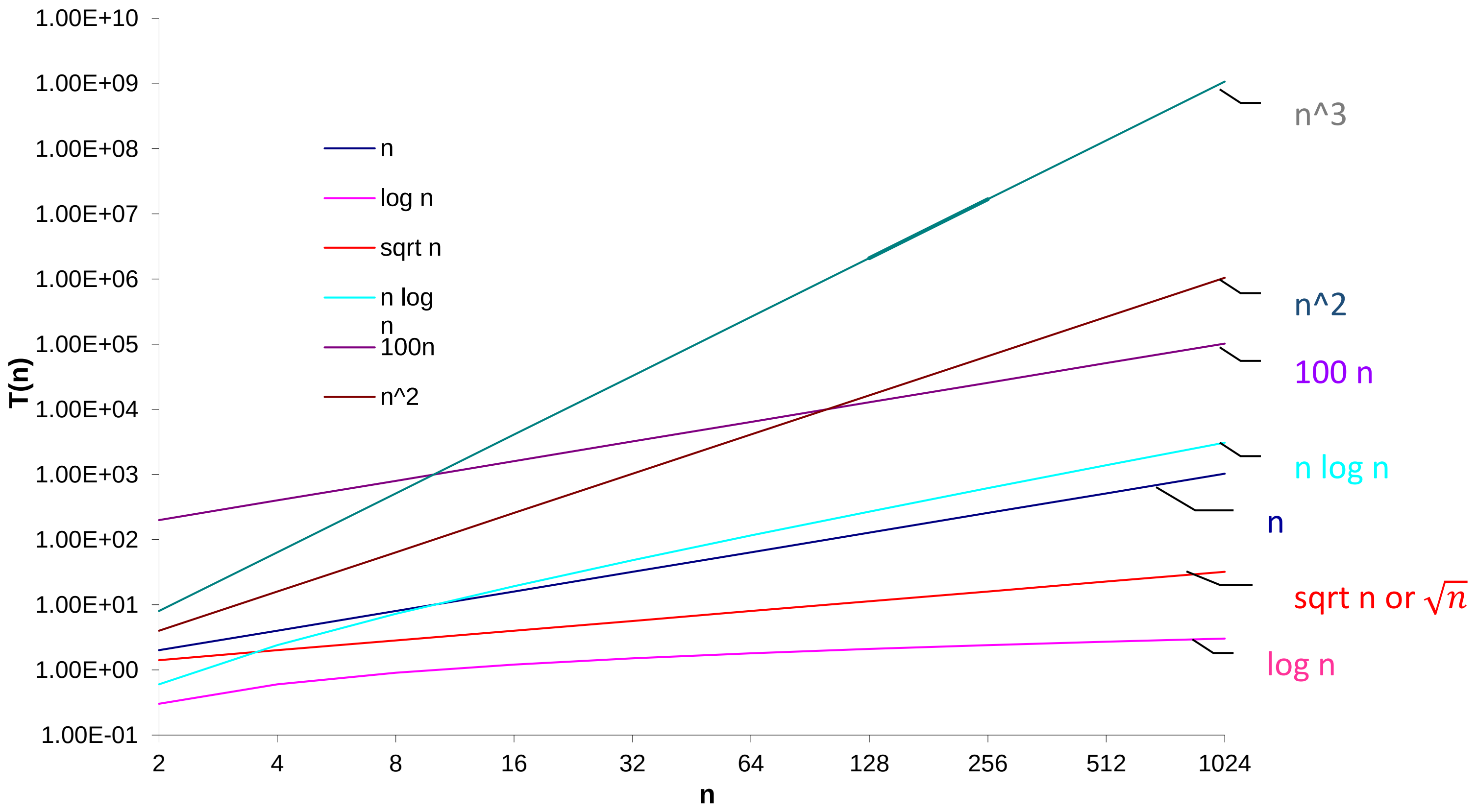
n	Log(n)	sqrt n	n	n log(n)	100n	n^2	n^3	2^n	n!
1	0	1	1	0	100	1	1	2	1
2	0.30103	1.41421 4	2	0.60206	200	4	8	4	2
3	0.477121	1.73205 1	3	1.431364	300	9	27	8	6
4	0.60206	2	4	2.40824	400	16	64	16	24
5	0.69897	2.23606 8	5	3.49485	500	25	125	32	120
6	0.778151	2.44949	6	4.668908	600	36	216	64	720
7	0.845098	2.64575 1	7	5.915686	700	49	343	128	5040
8	0.90309	2.82842 7	8	7.22472	800	64	512	256	40320
9	0.954243	3	9	8.588183	900	81	729	512	362880
10	1	3.16227 8	10	10	1000	100	1000	1024	3628800

Comparison on the number of data and the number of operations performed by Algorithm 1 and Algorithm 2.

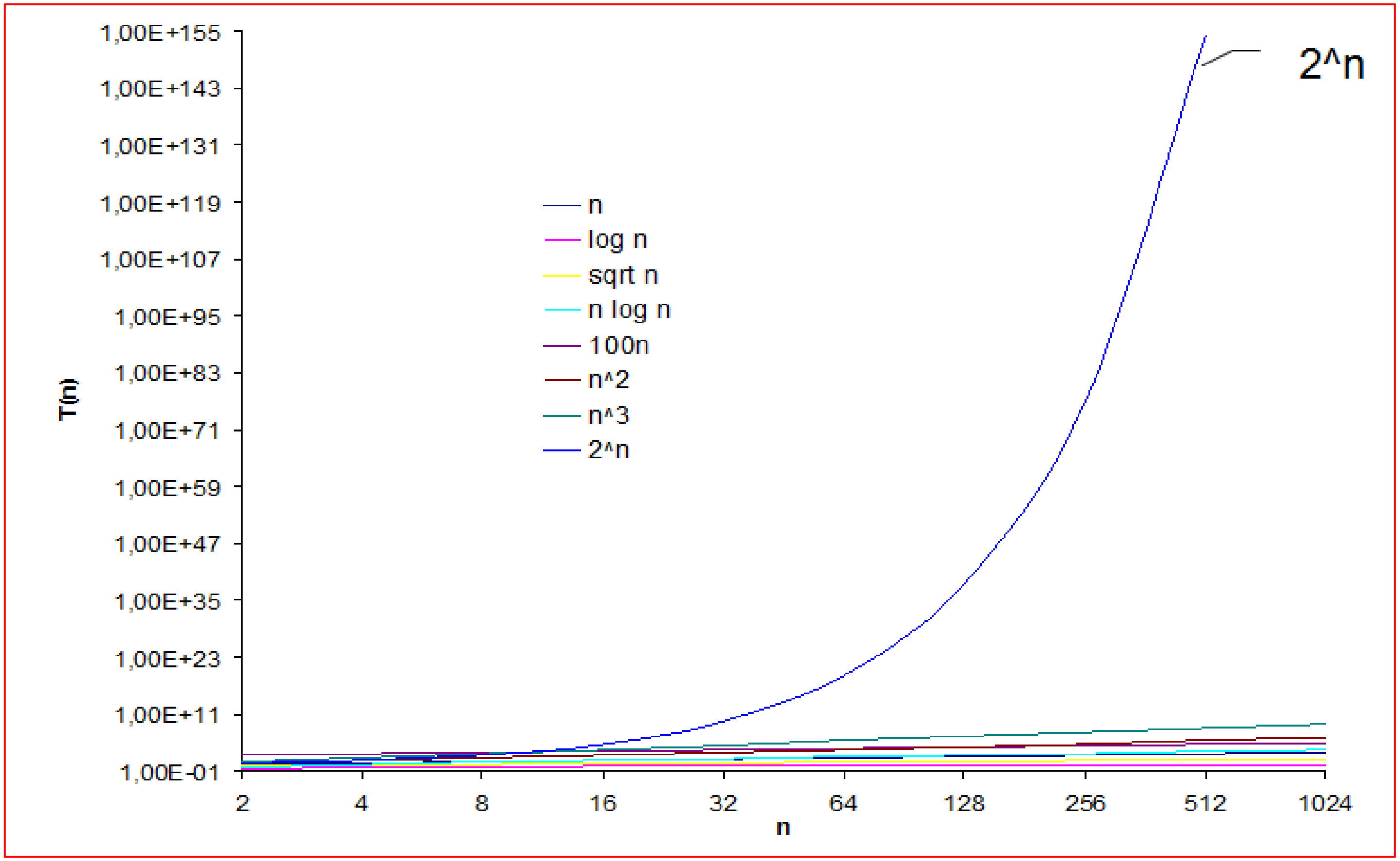
n	Algo 1 2n^2+7n+3	Algo 2 4n+58	$Ratio: \frac{Algo\ 1}{Algo\ 2}$
1	12	62	0.19
2	25	66	0.38
3	42	70	0.60
4	63	74	0.85
5	88	78	1.13
6	117	82	1.43
7	150	86	1.74
8	187	90	2.08
9	228	94	2.43
10	273	98	2.79



Classes of functions that are important for analysis of algorithms



Classes of functions that are important for analysis of algorithms



Classes of functions

- If we look at the running-time efficiency of Algorithm 1 and Algorithm 2 that we obtained earlier, Algorithm 1 has a **quadratic** complexity and Algorithm 2 has a **linear** complexity.

Algorithm 1:

$$\boldsymbol{T(n) = 2n^2 + 7n + 3}$$

Algorithm 2:

$$\boldsymbol{T(n) = 4n + 58}$$

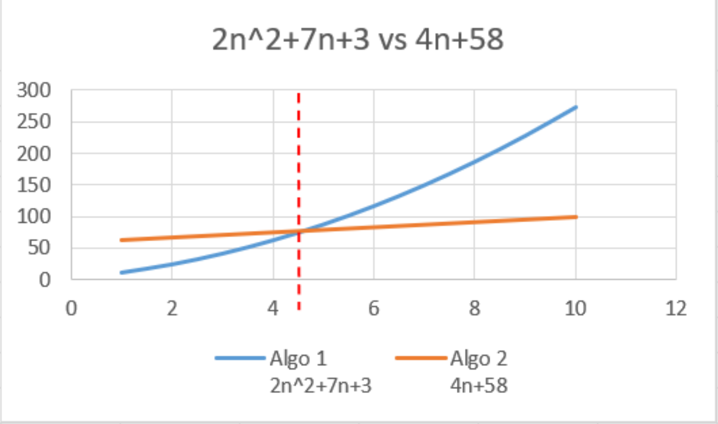
Classes of functions

- Instead of computing the running-time efficiency as what we did for Algorithm 1 and Algorithm 2, we can group algorithms into classes of functions, and based on the behaviour of the functions, we analyse and understand the complexity of the algorithm.
- We provide an abstract measure of this complexity by expressing it in terms of the problem size n.

Classes of functions

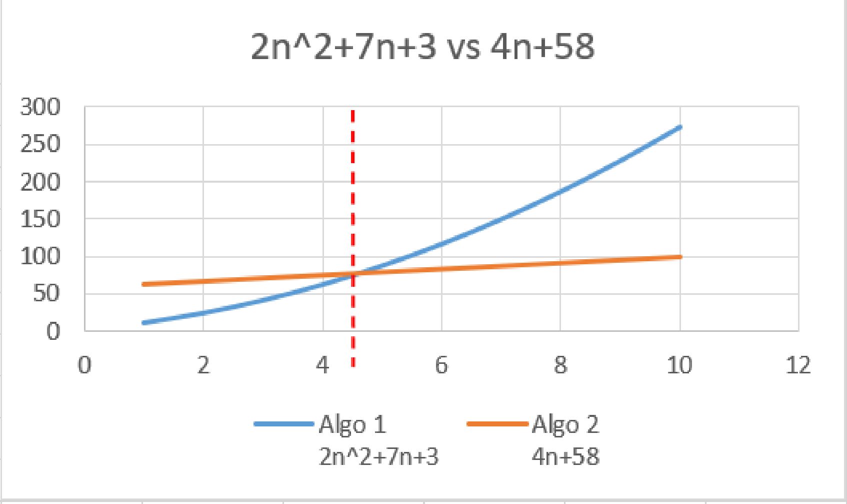
- A linear algorithm (Algorithm 2: $\boldsymbol{T(n) = 4n + 58}$) is always asymptotically better than a quadratic algorithm (Algorithm 1: $\boldsymbol{T(n) = 2n^2 + 7n + 3}$).
- This is because there is always some n at which the magnitude of $(2n^2 + 7n + 3)$ overtakes $(4n + 58)$.

n	Algo 1 2n^2+7n+3	Algo 2 4n+58	$Ratio: \frac{Algo\ 1}{Algo\ 2}$
1	12	62	0.19
2	25	66	0.38
3	42	70	0.60
4	63	74	0.85
5	88	78	1.13
6	117	82	1.43
7	150	86	1.74
8	187	90	2.08
9	228	94	2.43
10	273	98	2.79



Classes of functions

n	Algo 1 2n^2+7n+3	Algo 2 4n+58	$Ratio\colon \frac{Algo\ 1}{Algo\ 2}$
1	12	62	0.19
2	25	66	0.38
3	42	70	0.60
4	63	74	0.85
5	88	78	1.13
6	117	82	1.43
7	150	86	1.74
8	187	90	2.08
9	228	94	2.43
10	273	98	2.79



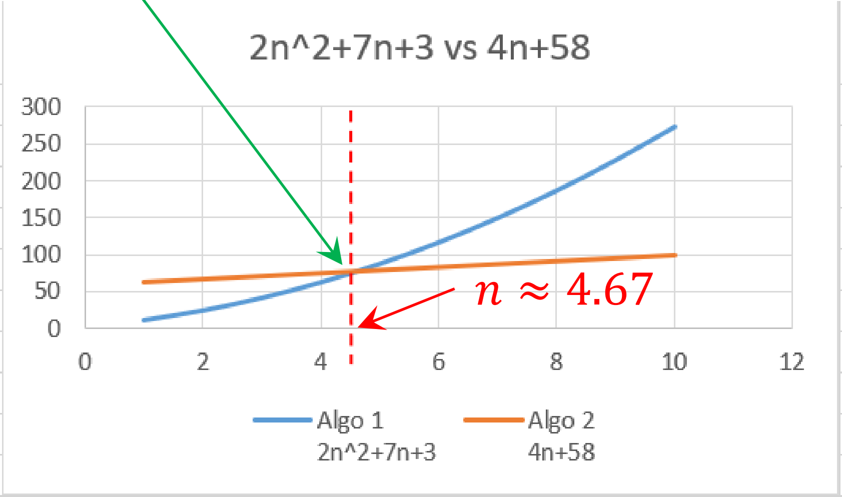
Quadratic vs Linear

We conclude that asymptotic behaviour has shown that the slower (lower) the growth rate as n (the number of data to be processed) increases, the better the performance of the algorithm.

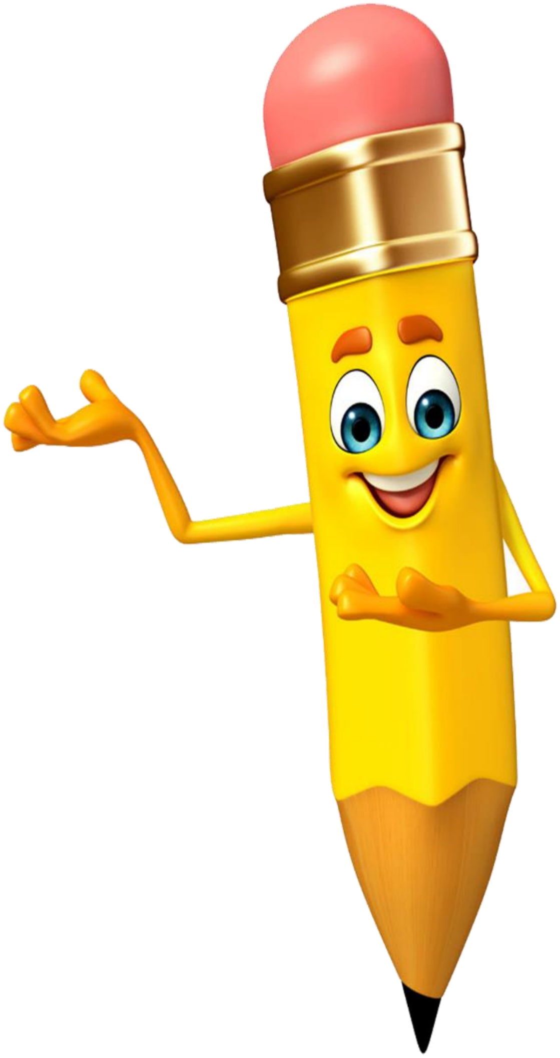
Classes of functions

n	Algo 1 2n^2+7n+3	Algo 2 4n+58	$Ratio{:}\frac{Algo\ 1}{Algo\ 2}$
1	12	62	0.19
2	25	66	0.38
3	42	70	0.60
4	63	74	0.85
5	88	78	1.13
6	117	82	1.43
7	150	86	1.74
8	187	90	2.08
9	228	94	2.43
10	273	98	2.79

$T(n) = 2n^2 + 7n + 3\ (Algo\ 1)$ overtakes $T(n) = 4n + 58\ (Algo\ 2)$ here.



Best-case, Average-case, and Worst-case



Worst-case, Best-case, and Average-case Efficiencies

- We understand the rule of thumb "the slower the asymptotic growth rate, the better the algorithm."
- However, this is often not the whole story.
- We have heard of the term **worst case** scenario in our earlier analysis of the two algorithms.
- We will use examples to explain what is **worst-case**, **best-case** and **average case** efficiencies mean next in order to have a better/clearer understanding of algorithm complexity.

Problem 1 (best-case efficiency)

- Given an array with the following content:

5	**3**	**4**	**1**	**2**	**0**	**8**	**6**	**7**	**9**

- What is the **first number** in the array?, and how many operation(s) need to be done to obtain the value?

Problem 1 (best-case efficiency)

- Given an array with the following content:

5	**3**	**4**	**1**	**2**	**0**	**8**	**6**	**7**	**9**

- What is the **first number** in the array?, and how many operation(s) need to be done to obtain the value?
- Of course the value is 5, and the number of operation to produce this output is **one**.

Problem 1 (best-case efficiency)

- Given an array with the following content:

5	**3**	**4**	**1**	**2**	**0**	**8**	**6**	**7**	**9**

- What is the **first number** in the array?, and how many operation(s) need to be done to obtain the value?
- Of course the value is 5, and the number of operation to produce this output is **one**.
- What happen if the content of the array is different?

Problem 1 (best-case efficiency)

- Given an array with the following content:

5	**3**	**4**	**1**	**2**	**0**	**8**	**6**	**7**	**9**

- What is the **first number** in the array?, and how many operation(s) need to be done to obtain the value?
- Of course the value is 5, and the number of operation to produce this output is **one**.
- What happen if the content of the array is different?
- It does not matter, the number of operation to produce the output is still one.

Problem 1 (best-case efficiency)

- In this example, the number of operations performed to obtain the required result is always 1, regardless of the data. This behaviour is known as **best-case scenario** of running-time efficiency of an algorithm.
- The **best-case** efficiency of an algorithm is its efficiency for the best-case input of size n, which is an input of size n for which the algorithm runs the fastest (least number of operations performed) among all possible inputs of that size.

> Note: The best case does not mean the smallest input; it means with the input of size n for which the algorithm runs the fastest.

Problem 2 (worst-case efficiency)

- Given an array with the following content:

5	**3**	**4**	**1**	**2**	**0**	**8**	**6**	**7**	**9**

- In which array location the number 18 can be found?
- Of course none of the 10 locations contain the number 18.
- In this case, one way to find out if the number 18 exists in the array is by comparing (checking) the content of the array from the first element until the last element, in this case n (the size of the data set.)

Problem 2 (worst-case efficiency)

- The running-time efficiency of the algorithm used to solve problem 2 is known as **worst-case** efficiency. (Unsuccessful search of an item from an arrays of n elements using sequential search.)
- The **worst-case** efficiency of an algorithm is its efficiency for the worst-case input of size n, that is, an input (or inputs) of size n for which the algorithm runs the longest among all possible inputs of that size.

Problem 2 (worst-case efficiency)

- The running-time efficiency of the algorithm used to solve problem 2 is known as **worst-case** efficiency.
- The **worst-case** efficiency of an algorithm is its efficiency for the worst-case input of size n, which is an input (or inputs) of size n for which the algorithm runs the longest among all possible inputs of that size.

The term 'longest' here means the most (largest) number of operations perform to obtain the result. The worst case does not mean the number of operations equals to the number of data n.

Problem 3 (average-case efficiency)

- Given an array with the following content:

5	**3**	**4**	**1**	**2**	**0**	**8**	**6**	**7**	**9**

- In which array location the number 8 can be found?
- By checking the element one-by-one from the beginning until we finally find the number 8 in location 6.
- But what happen if the content of the array changes?
- The algorithm will not change, that is, we still check the element one-by-one from the beginning until we find the number we want.

Problem 3 (average-case efficiency)

- Of course the number that we want to find can be in any location of the array.
- The running-time efficiency of the algorithm used to solve problem 3 is known as **average-case** efficiency.
- The analysis of average-case efficiency is generally harder than best-case and worst-case efficiencies, since it involves probabilistic component and often requires assumptions about the distribution of inputs.

Problem 3 (average-case efficiency)

- In determining the average-case efficiency of an algorithm, the standard assumptions are that:
  1. The probability of a successful search is equal to p, where $0 \leq p \leq 1$.
  2. The probability of a successful search for every numbers in the array is the same, that is, all numbers in the array have an equal chance of successfully found.

Problem 3 (average-case efficiency)

- The average number of key comparison $C_{avg}(n)$ is as follows:
  - In the case of successful search, the probability of the first match occurring in the i^{th} position of the list is $\frac{p}{n}$ for every i, and the number of comparisons made by the algorithm, in such a situation, is obviously i.
  - In the case of unsuccessful search, the number of comparisons will be n with the probability of such a search being $(1 - p)$.

Problem 3 (average-case efficiency)

- Hence,

$$C_{avg}(n) = \left[1 \times \frac{p}{n} + 2 \times \frac{p}{n} + \cdots + i \times \frac{p}{n} + \cdots + n \times \frac{p}{n}\right] + n \times (1 - p)$$

$$= \frac{p}{n}\ [1 + 2 + \cdots + i + \cdots + n] + n(1 - p)$$

$$= \frac{p}{n}\left(\frac{n \times (n + 1)}{2}\right) + n(1 - p)$$

$$= \frac{p(n + 1)}{2} + n(1 - p)$$

Problem 3 (average-case efficiency)

- The average-case efficiency of an algorithm is its efficiency for the average-case input of size n, which is an input (or inputs) of size n for which the algorithm will inspect, on average, $C_{avg}(n) = \frac{p(n+1)}{2} + n(1-p)$ number of key comparisons.

$C_{avg}(n) = \frac{p(n+1)}{2} + n(1-p)$

When p=1 (the search must be successful), $C_{avg}(n) = \frac{(n+1)}{2}$.

When p=0 (the search must be unsuccessful), $C_{avg}(n) = n$.

Will the **best-case** run-time complexity and the **worst-case** run-time complexity of an algorithm be the same?

Will the **best-case** run-time complexity and the **worst-case** run-time complexity of an algorithm be the same?

So far we have seen examples on the best-case, worst-case, and average-case. In all the three examples, the best-case, worst-case, and average-case complexity is affected by the content of the data.

Problem 4 (Finding Maximum)

- One example of an algorithm whose running times growth only changes with the input data size and not with the content of the data is **find maximum**.

$$
\begin{array}{l}
\mathit{Function}\ \max(A) \\
\quad \max = A[0] \\
\quad \mathit{for}\ 0 \le i < n \\
\quad\quad \mathit{if}\ (A[i] > \max) \\
\quad\quad\quad \max = A[i] \\
\quad \mathit{return}\ \max
\end{array}
$$

Problem 4 (Finding Maximum)

$$
\begin{array}{l}
\mathit{Function}\ \max(A) \\
\quad \max = A[0] \\
\quad \mathit{for}\ 0 \leq i < \ n \\
\quad\quad \mathit{if}\ (A[i] > \max) \\
\quad\quad\quad \max = A[i] \\
\quad \mathit{return}\ \max
\end{array}
$$

- This algorithm returns the position of the element with the highest value in the array A.

0	1	2	3	4	5	6	7	
9	6	8	2	3	7	5	1	Max = 9

0	1	2	3	4	5	6	7	
3	6	1	5	7	2	4	8	Max = 8

Problem 4 (Finding Maximum)

- With the example finding maximum above, the algorithm always execute n number of times regardless of whether the maximum values is in the first position, somewhere in the middle, or in the last position.
- In this case, the worst-case and the best case running time of the algorithm is the same.

Complexity Analysis

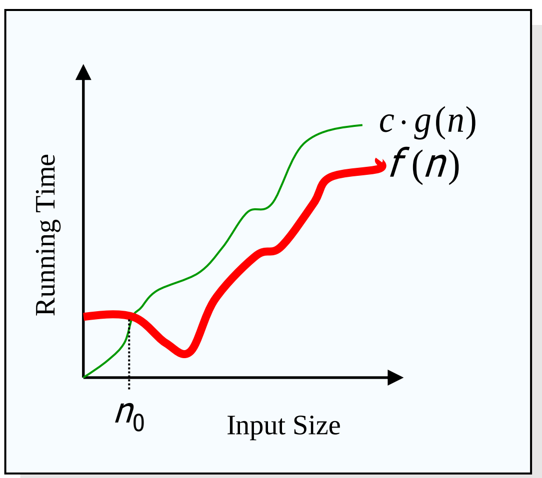
Asymptotic Notation (ΟΩΘ)

Asymptotic Notation

- Now that we have known how to analyse and determine the running-time efficiency of algorithms, how do we compare and ranks algorithm based on order of complexity (asymptotically)?
- There are three standard order of complexity measures in common use:
  - Big Oh (O)
  - Big Omega (Ω)
  - Big Theta (Θ)

Big-O (Big O)

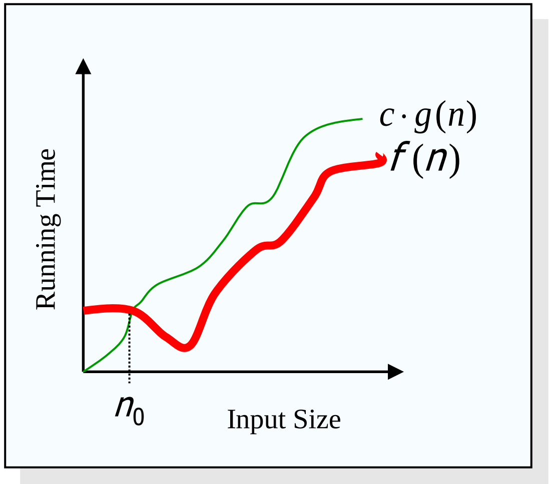
- We use big-O notation to asymptotically bound the growth of a running-time of a function ($f(n)$) to **lower** or **same** order of growth of another function ($g(n)$) within a constant factor.
- From the run-time expression that we obtained during the analysis, we plot the expression as a function of n.



- In the diagram shown here, the horizontal axis represents the input data size, and the vertical axis represents the computational requirements of an algorithm.

Big-O (Big O)

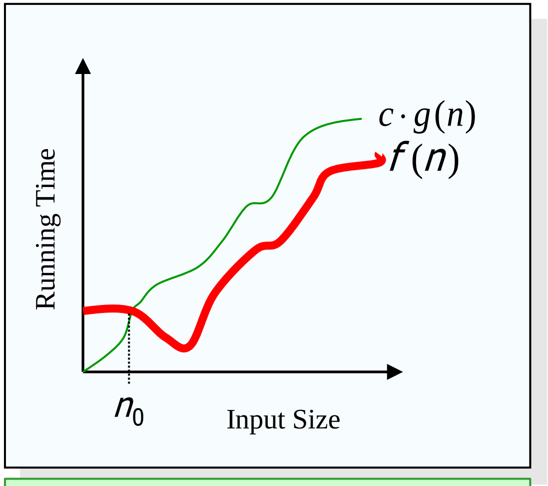
- Our aim is to find a function g of n that acts as an upper bound for f of n.
- It does not matter that g of n is less than f of n for the lower values of n. What is important is that for large values of n, g of n is always equal to or higher than f of n.



Big-O (Big O)

- Formal definition:

Given non-negative functions $f(n)$ and $g(n)$, we say that $f(n) \in O(g(n))$, if there exists an integer n_0 and a constant $c > 0$ such that $f(n) \leq cg(n)$ for all integers $n \geq n_0$.

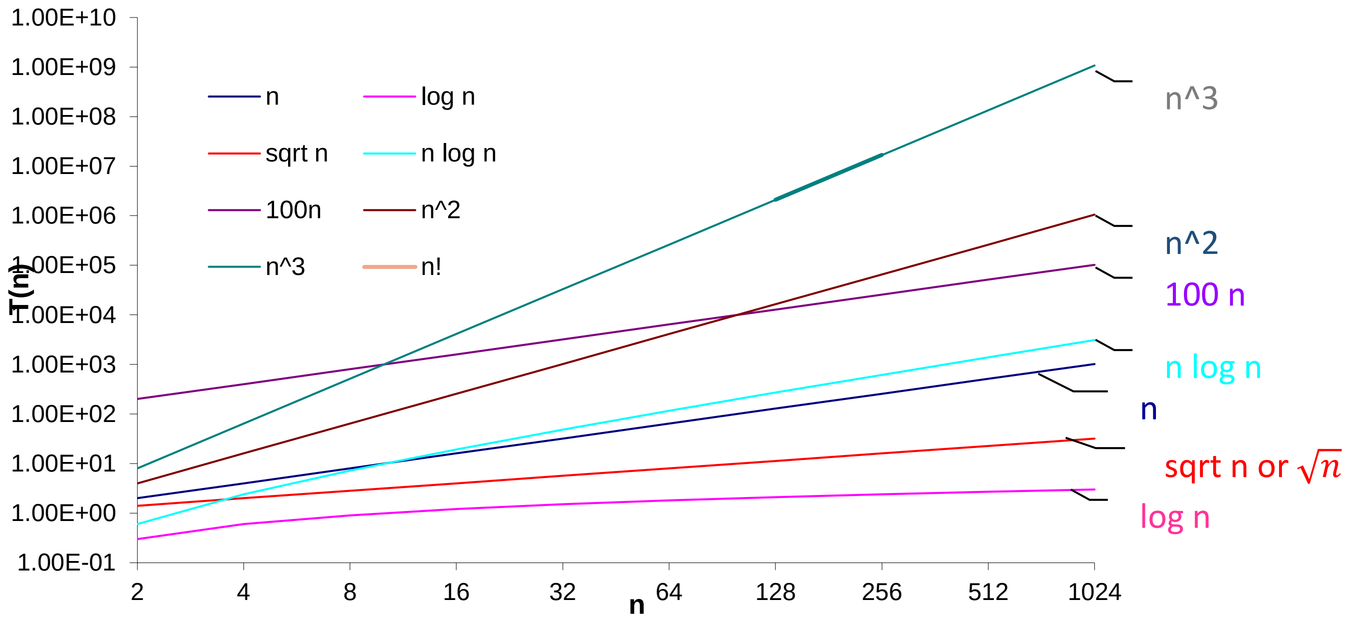


$f(n)$ is **bounded above** by c. $g(n)$.

$f(n) \in O(g(n))$: $f(n)$ is of order **at most** $g(n)$ or $f(n)$ is Big-O of $g(n)$.

Big-O (Big O)

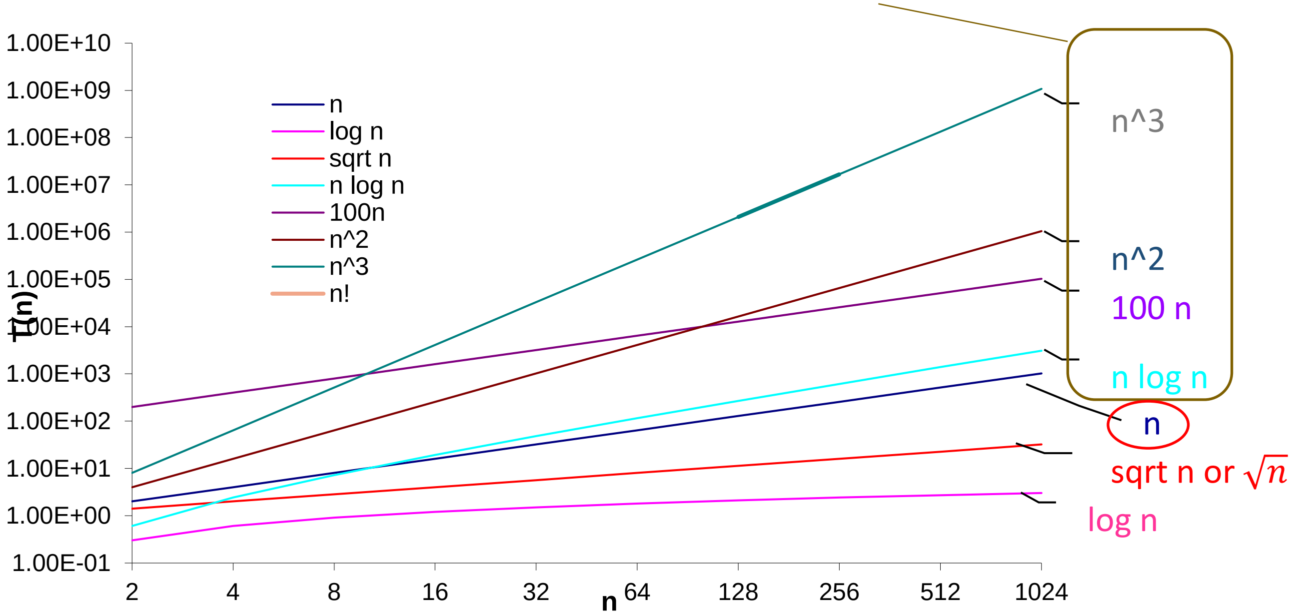
- By now, you might have already realized that there is a set of functions that can act as an upper bound for $f(n)$.



Big-O (Big O)

- By now, you might have already realized that there is a set of functions that can act as an upper bound for $f(n)$.

$f(n)$ is $O(n \lg n), O(100n), O(n^2), and\ O(n^3)$.



Big-O (Big O)

Examples:

- $n \in O(n^2)$
- $100n + 5 \in O(n^2)$
- $\frac{1}{2}n(n-1) \in O(n^2)$

Big-O (Big O)

Example: $n \in O(n^2)$

Proof:

We need to find n_0 and c such that

$f(n) \leq cg(n)$ for all $n \geq n_0$.

Big-O (Big O)

Example: $n \in O(n^2)$

Proof:

We need to find n_0 and c such that

$f(n) \leq cg(n)$ for all $n \geq n_0$.

- Let $f(n) = n$, and $g(n) = n^2$.
- $We\ have$: $n \leq cn^2$
- $\rightarrow \frac{n}{n^2} \leq c$
- $\rightarrow \frac{1}{n} \leq c$

Big-O (Big O)

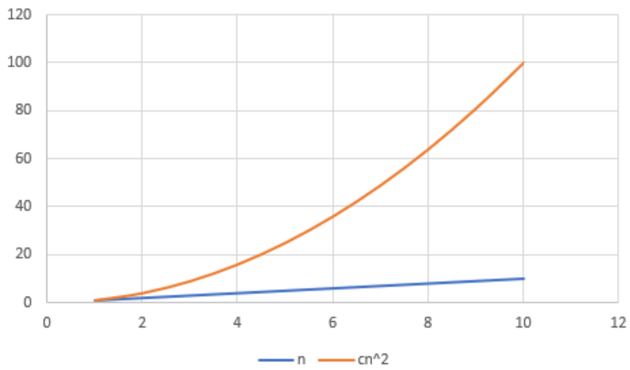
Example: $n \in O(n^2)$

Proof:

We need to find n_0 and c such that

$f(n) \leq cg(n)$ for all $n \geq n_0$.

- Let $f(n) = n$, and $g(n) = n^2$.
- $We\ have$: $n \leq cn^2$
- $\rightarrow \frac{n}{n^2} \leq c$
- $\rightarrow \frac{1}{n} \leq c$



Pick $n_0 = 1$ and $c = 1$ and the inequality stands for all value of $n \geq n_0$ and $c = 1$. Hence, the statement $n \in O(n^2)$ is true. The proof completes.

Big-O (Big O)

Example: $100n + 5 \in O(n^2)$

Proof:

We need to find n_0 and c such that

$f(n) \leq cg(n)$ for all $n \geq n_0$.

Big-O (Big O)

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Example: $100n + 5 \in O(n^2)$

Proof:

We need to find n_0 and c such that

$f(n) \leq cg(n)$ for all $n \geq n_0$.

- Let $f(n) = 100n + 5$ and $g(n) = n^2$.
- $\rightarrow 100n + 5 \leq cn^2$
- $\rightarrow cn^2 - 100n \geq 5$

Big-O (Big O)

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

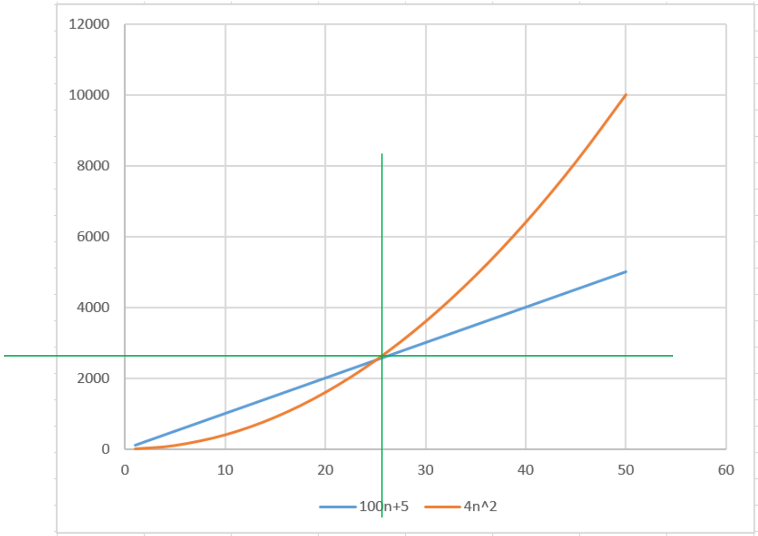
Example: $100n + 5 \in O(n^2)$

Proof:

We need to find n_0 and c such that

$f(n) \leq cg(n)$ for all $n \geq n_0$.

- Let $f(n) = 100n + 5$ and $g(n) = n^2$.
- $\rightarrow 100n + 5 \leq cn^2$
- $\rightarrow cn^2 - 100n \geq 5$



Pick $n_0 \approx 25.05$ and $c = 4$ and the inequality stands for all value of $n \geq n_0$ and $c = 4$. Hence, the statement $100n + 5 \in O(n^2)$ is true. The proof completes

Big-O (Big O)

Example: $\frac{1}{2}n(n-1) \in O(n^2)$

Proof:

We need to find n_0 and c such that

$f(n) \leq cg(n)$ for all $n \geq n_0$.

Big-O (Big O)

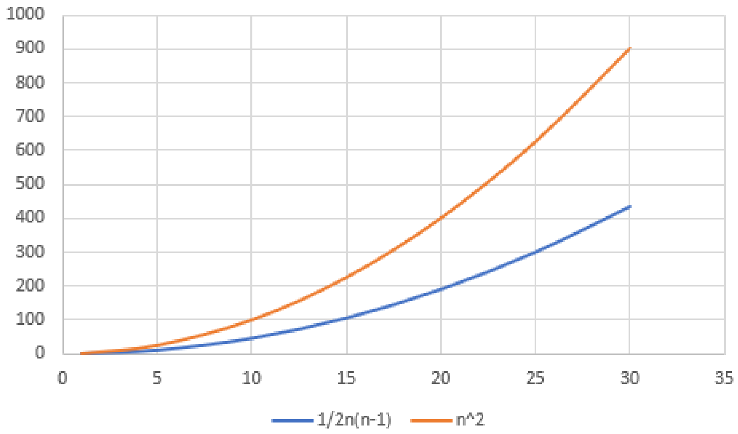
Example: $\frac{1}{2}n(n-1) \in O(n^2)$

Proof:

We need to find n_0 and c such that

$f(n) \leq cg(n)$ for all $n \geq n_0$.

- Let $f(n) = \frac{1}{2}n(n-1)$ and
  $g(n) = n^2$.
- $\frac{1}{2}n(n-1) \leq cn^2$
- $\frac{1}{2}n^2 - \frac{1}{2}n \leq cn^2$
- $\mathrm{c}n^2 - \frac{1}{2}n^2 + \frac{1}{2}n \geq 0$
- $\left(c - \frac{1}{2}\right)n^2 + \frac{1}{2}n \geq 0$



Big-O (Big O)

Example: $\frac{1}{2}n(n-1) \in O(n^2)$

Proof:

We need to find n_0 and c such that $f(n) \leq cg(n)$ for all $n \geq n_0$.

- Let $f(n) = \frac{1}{2}n(n-1)$ and $g(n) = n^2$.
- $\frac{1}{2}n(n-1) \leq cn^2$
- $\frac{1}{2}n^2 - \frac{1}{2}n \leq cn^2$
- $cn^2 - \frac{1}{2}n^2 + \frac{1}{2}n \geq 0$
- $\left(c - \frac{1}{2}\right)n^2 + \frac{1}{2}n \geq 0$

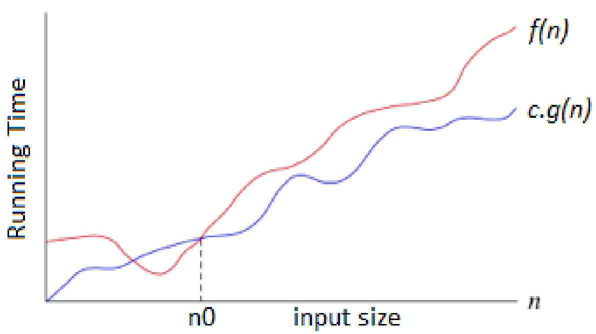
Pick $n_0 = 1$ and $c = 1$ and the inequality stands for all values of $n \geq n_0$ and $c = 1$. Hence, the statement $\frac{1}{2}n(n-1) \in O(n^2)$ is valid. Proof completes.

Note: The value of c must be greater or equal to 0.5. This is to ensure we are working with a positive (non-negative) functions.

Big-Ω (Big Omega)

- We use big-Ω notation to asymptotically bound the growth of a running-time of a function to **higher** or **same** order of growth of another function within a constant factor.

Big Ω (Big Omega)



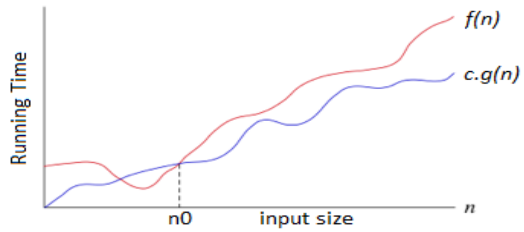
- If you were able to understand the Big O notation, understanding the Big Omega notation should be easy, because it is analogous to the Big O notation.

Instead of looking for functions that act as an upper bound for the running time of algorithm, we will be looking for functions that act as a lower bound.

Big-Ω (Big Omega)

- Formal definition:

Given non-negative functions $f(n)$ and $g(n)$, we say that $f(n) \in \Omega(g(n))$, if there exists an integer n_0 and a constant $c > 0$ such that $f(n) \geq cg(n)$ for all integers $n \geq n_0$.

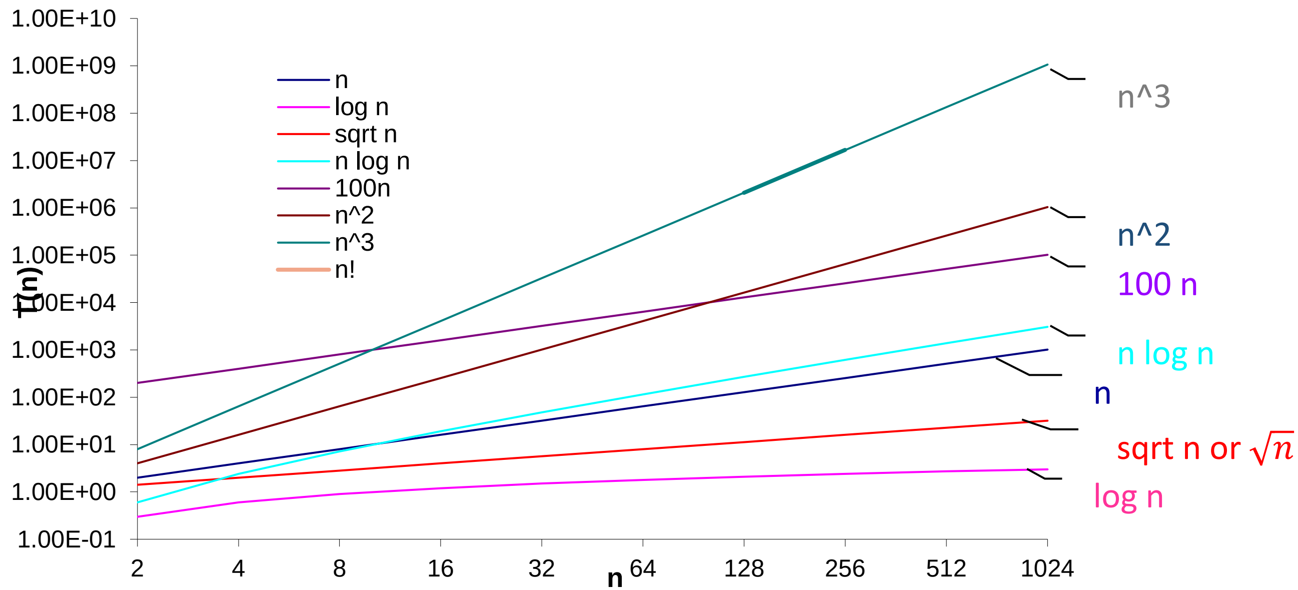


$f(n)$ is **bounded below** by c. $g(n)$.

$f(n) \in \Omega\big(g(n)\big)$: $f(n)$ is of order **at least** $g(n)$ or $f(n)$ is Big-Ω of $g(n)$.

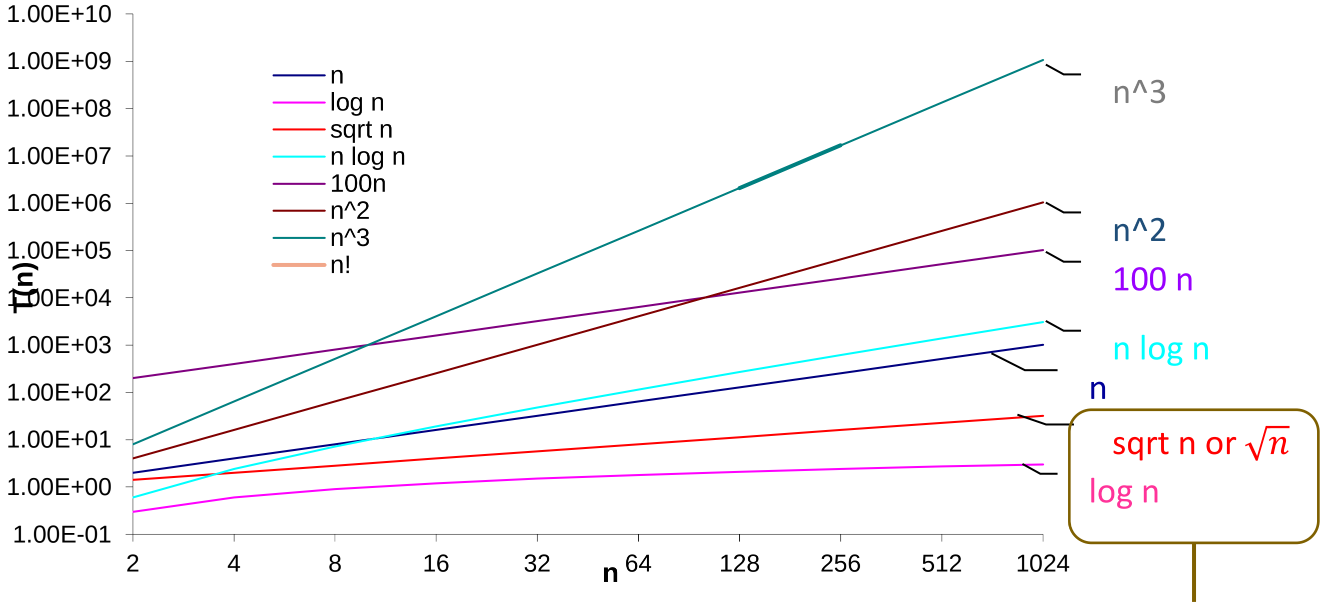
Big-Ω (Big Omega)

- Contrary to the Big O, there is a set of functions that can act as an lower bound for $f(n)$.



Big-Ω (Big Omega)

- Contrary to the Big O, there is a set of functions that can act as an lower bound for $f(n)$.



$f(n)$ is $\Omega(\sqrt{n})$ and $f(n)$ is $\Omega(\log n)$

Big-Ω (Big Omega)

Examples:

- $7n^2 + n \in \Omega(n^2)$
- $8n + 128 \in \Omega(n)$
- $n^3 \in \Omega(n^2)$

I will show the proof for the first and second examples. I leave third example for you to practice.

Big-Ω (Big Omega)

Example: $7n^2 + n \in \Omega(n^2)$

Proof:

We need to find n_0 and c such that

$f(n) \geq cg(n)$ for all $n \geq n_0$.

Big-Ω (Big Omega)

Example: $7n^2 + n \in \Omega(n^2)$

Proof:

We need to find n_0 and c such that

$f(n) \geq cg(n)$ for all $n \geq n_0$.

- Let $f(n) = 7n^2 + n$, and $g(n) = n^2$.
- $\rightarrow 7n^2 + n \geq cn^2$
- $\rightarrow 7n^2 - cn^2 + n \geq 0$

Big-Ω (Big Omega)

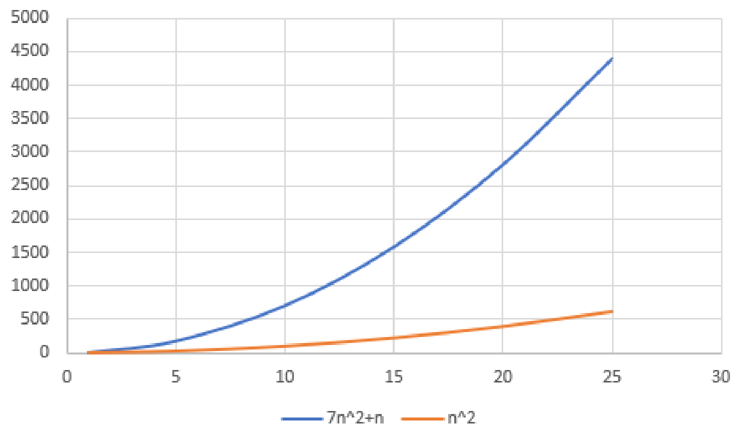
Example: $7n^2 + n \in \Omega(n^2)$

Proof:

We need to find n_0 and c such that

$f(n) \geq cg(n)$ for all $n \geq n_0$.

- Let $f(n) = 7n^2 + n$, and $g(n) = n^2$.
- $\rightarrow 7n^2 + n \geq cn^2$
- $\rightarrow 7n^2 - cn^2 + n \geq 0$



Pick $n_0 = 1$ and $c = 1$ and the inequality stands for all values of $n \geq n_0$ and $c = 1$. Hence, the statement $7n^2 + n \in \Omega(n^2)$ is valid. The proof completes.

Big-Ω (Big Omega)

Example: $8\mathrm{n} + 128 \in \Omega(n)$

Proof:

We need to find n_0 and c such that

$f(n) \geq cg(n)$ for all $n \geq n_0$.

Big-Ω (Big Omega)

Example: $8\text{n} + 128 \in \Omega(n)$

Proof:

We need to find n_0 and c such that

$f(n) \geq cg(n)$ for all $n \geq n_0$.

- Let $f(n) = 8n + 128$, and $g(n) = n$.
- $\rightarrow 8n + 128 \geq cn$
- $\rightarrow 8n - cn + 128 \geq 0$
- $\rightarrow (8 - c)n + 128 \geq 0$

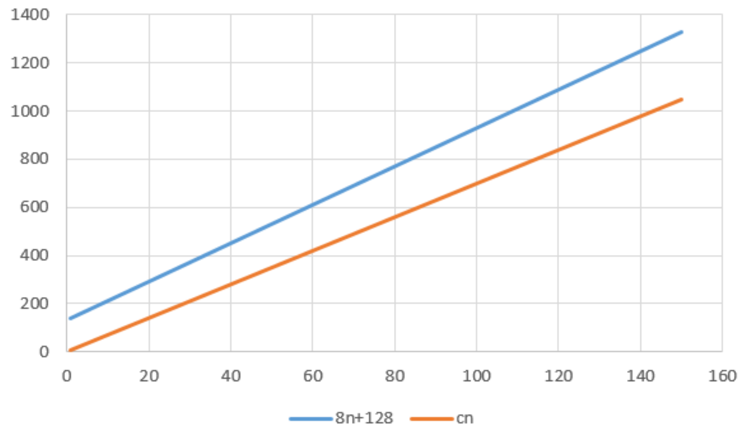
Big-Ω (Big Omega)

Example: $8\text{n} + 128 \in \Omega(n)$

Proof:

We need to find n_0 and c such that $f(n) \geq cg(n)$ for all $n \geq n_0$.

- Let $f(n) = 8n + 128$, and $g(n) = n$.
- $\leftrightarrow 8n + 128 \geq cn$
- $\leftrightarrow 8n - cn + 128 \geq 0$
- $\leftrightarrow (8 - c)n + 128 \geq 0$

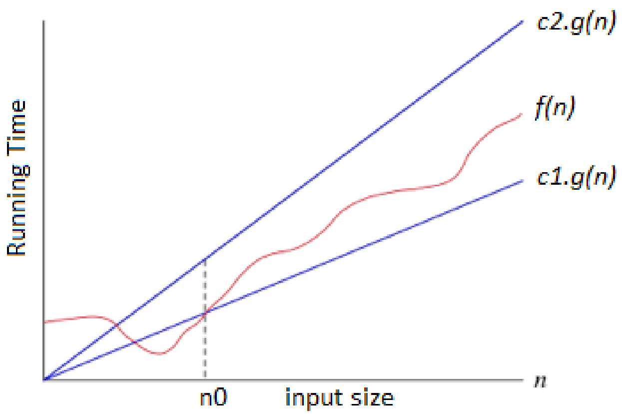


Pick $n_0 = 1$ and $c = 7$ and the inequality stands for all values of $n \geq n_0$ and $c = 7$. Hence, the statement $8n + 128 \in \Omega(n)$ is valid. The proof completes.

Big-Θ (Big Theta)

- We use big-Θ notation to asymptotically bound the growth of a running time to within constant factors **above** and **below** or the **same** order of growth of another function within the bound.
- One drawback of the Big O and Big Ω notations is that they refer to a set of functions.
- Theta notation finds a single function that acts simultaneously as an upper- and a lower-bound for our running time function or memory space function, which has a more meaningful representation.

Big-Θ (Big Theta)



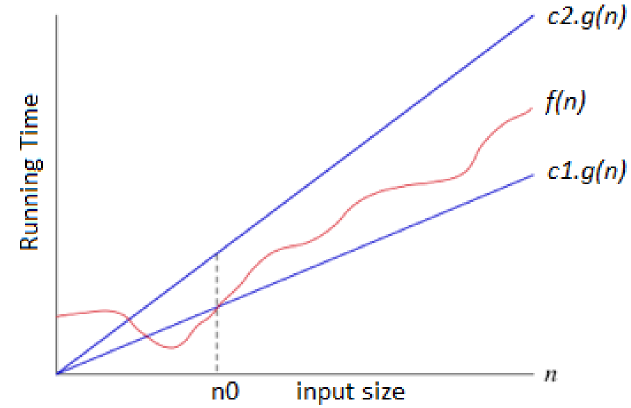
With Theta notation, our aim is to find a function $g(n)$ that acts simultaneously as an upper and a lower bound for $f(n)$, depending on the value of the constant it is multiplied by.

Hence, if $g(n)$ is multiplied by $c1$, it acts as a lower bound for $f(n)$. But if $g(n)$ is multiplied by $c2$, and $c2$ is greater than $c1$, then it acts as an upper bound for $f(n)$.

Big-Θ (Big Theta)

- Formal definition:

Given non-negative functions $f(n)$ and $g(n)$, we say that $f(n) \in \Theta(g(n))$, if there exists an integer n_0 and a constant $c > 0$ such that $c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all integers $n \geq n_0$.



$f(n)$ is **bounded below** by $c_1.g(n)$ and **above** by $c_2.g(n)$.

$f(n) \in \Theta\big(g(n)\big)$: $f(n)$ is of **tightly ordered** by $g(n)$ or $f(n)$ is Big-Θ of $g(n)$ if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

Big-Θ (Big Theta)

Examples:

- $\frac{1}{2}n(n-1) \in \Theta(n^2)$
- $60n^2 + 50n + 1 \in \Theta(n^2)$
- $2n + 3\lg n \in \Theta(n)$

Big-Θ (Big Theta)

Example: $\frac{1}{2}n(n-1) \in \Theta(n^2)$

Proof:

We need to find n_0 and c such that

$f(n) \geq c_1 g(n)$ and $f(n) \leq c_2 g(n)$ for all $n \geq n_0$.

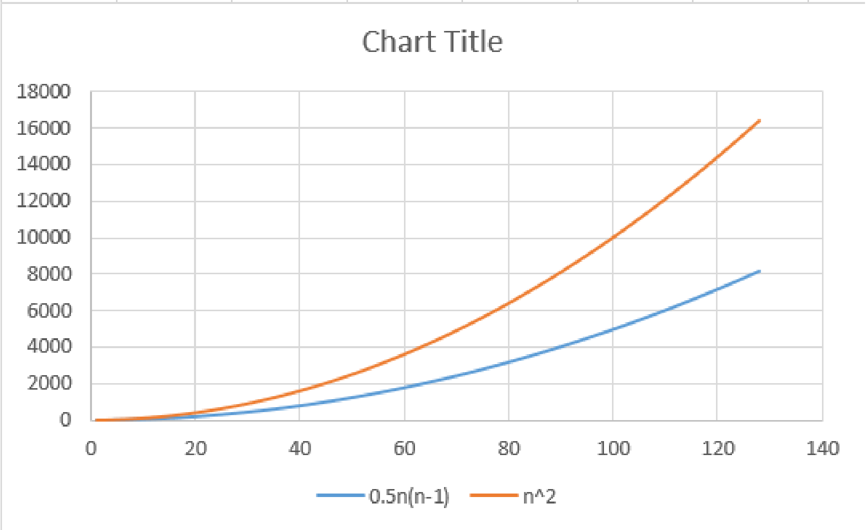
- Let $f(n) = \frac{1}{2}n(n-1)$, and $g(n) = n^2$.
- First we prove that $f(n) \in O(g(n))$
- Then we prove that $f(n) \in \Omega(g(n))$

Big-Θ (Big Theta)

- Prove that $f(n) \in O(g(n))$
- $\rightarrow f(n) \leq cg(n)$
- $\rightarrow \frac{1}{2}n(n-1) \leq cn^2$
- $\rightarrow \frac{1}{2}n^2 - \frac{1}{2}n \leq cn^2$
- $\rightarrow cn^2 - \frac{1}{2}n^2 + \frac{1}{2}n \geq 0$

Big-Θ (Big Theta)

- Prove that $f(n) \in O(g(n))$
- $\rightarrow f(n) \leq cg(n)$
- $\rightarrow \frac{1}{2}n(n-1) \leq cn^2$
- $\rightarrow \frac{1}{2}n^2 - \frac{1}{2}n \leq cn^2$
- $\rightarrow cn^2 - \frac{1}{2}n^2 + \frac{1}{2}n \geq 0$
- $\rightarrow \left(c - \frac{1}{2}\right)n^2 + \frac{1}{2}n \geq 0$



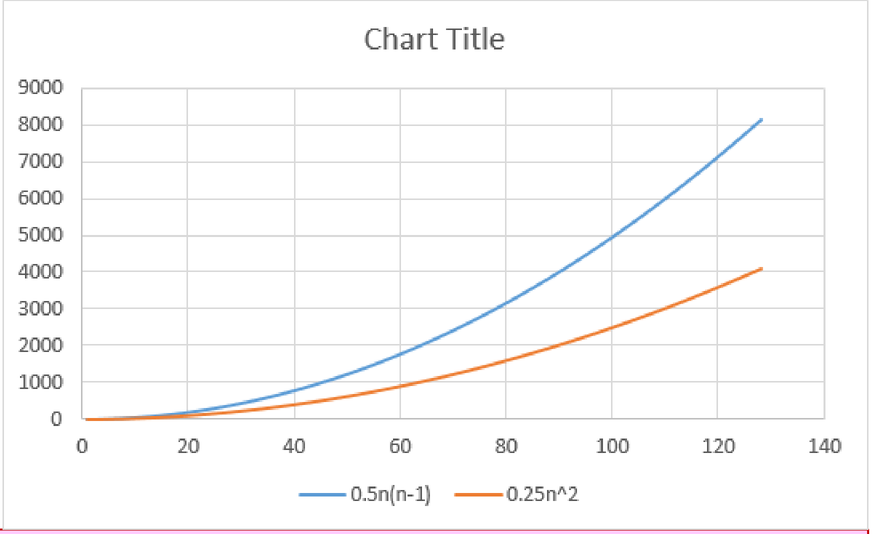
Pick $n_0 = 1$ and $c = 1$ and the inequality stands for all value of $n \geq n_0$ and $c = 1$. Hence, the proof completes.

Big-Θ (Big Theta)

- Prove that $f(n) \in \Omega(g(n))$
- $\rightarrow f(n) \geq cg(n)$
- $\rightarrow \frac{1}{2}n(n-1) \geq cn^2$
- $\rightarrow \frac{1}{2}n^2 - \frac{1}{2}n \geq cn^2$
- $\rightarrow \frac{1}{2}n^2 - cn^2 - \frac{1}{2}n \geq 0$

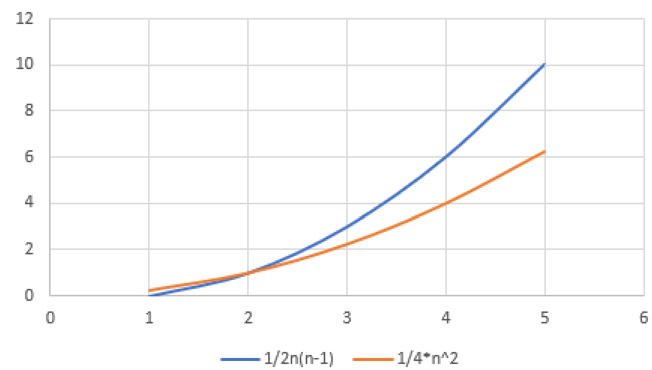
Big-Θ (Big Theta)

- Prove that $f(n) \in \Omega(g(n))$
- $\leftrightarrow f(n) \geq cg(n)$
- $\leftrightarrow \frac{1}{2}n(n-1) \geq cn^2$
- $\leftrightarrow \frac{1}{2}n^2 - \frac{1}{2}n \geq cn^2$
- $\leftrightarrow \frac{1}{2}n^2 - cn^2 - \frac{1}{2}n \geq 0$
- $\rightarrow \left(\frac{1}{2} - c\right)n^2 - \frac{1}{2}n \geq 0$

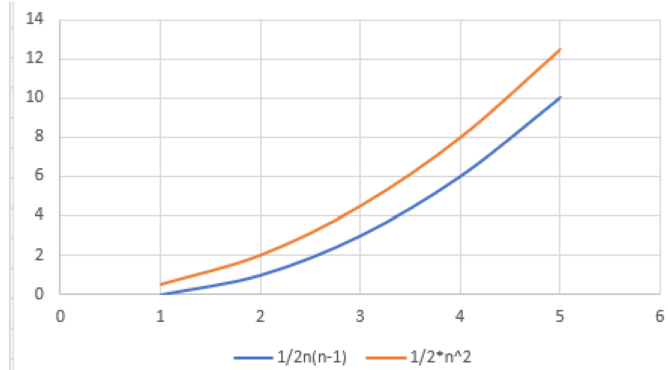


Pick $n_0 = 2$ and $c = \frac{1}{4}$ and the inequality stands for all values of $n \geq n_0$ and $c = \frac{1}{4}$. Hence the statement $f(n) \in \Omega(g(n))$ is valid. The proof completes.

Big-Θ (Big Theta)



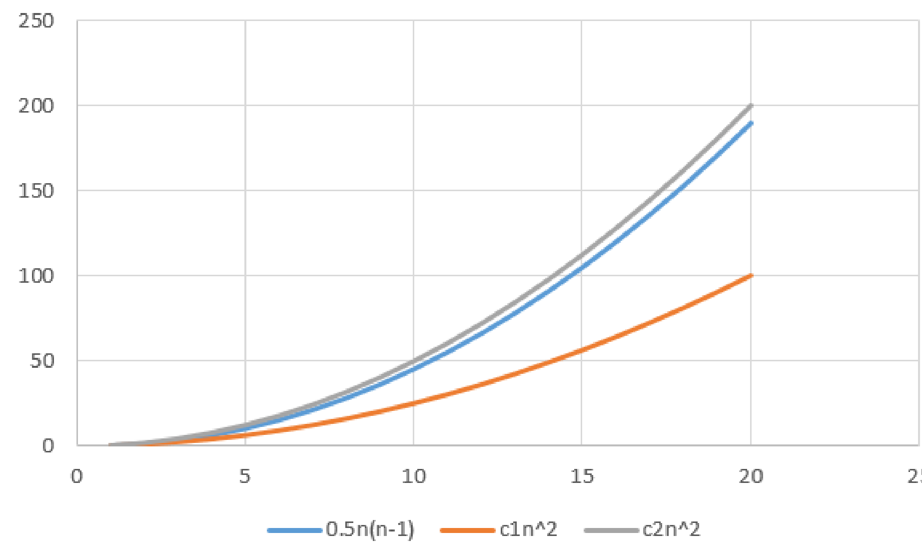
1/2n(n-1) >= 1/4*n^2



1/2n(n-1) <= 1/2*n^2

$$\frac{1}{2}n(n-1) \in \Theta(n^2)$$

Hence, $f(n) \in \Omega(g(n))$ for some constant $c_1 = \frac{1}{4}$ and $n_0 = 2$; and $f(n) \in O(g(n))$ for some constant $c_2 = \frac{1}{2}$ and $n_0 = 1$.



Big-Θ (Big Theta)

Example: $60n^2 + 50n + 1 \in \Theta(n^2)$

Proof:

We need to find n_0 and c such that

$f(n) \geq c_1 g(n)$ and $f(n) \leq c_2 g(n)$ for all $n \geq n_0$.

- Let $f(n) = 60n^2 + 50n + 1$, and $g(n) = n^2$.

Big-Θ (Big Theta)

Example: $2n + 3\lg n \in \Theta(n)$

Proof:

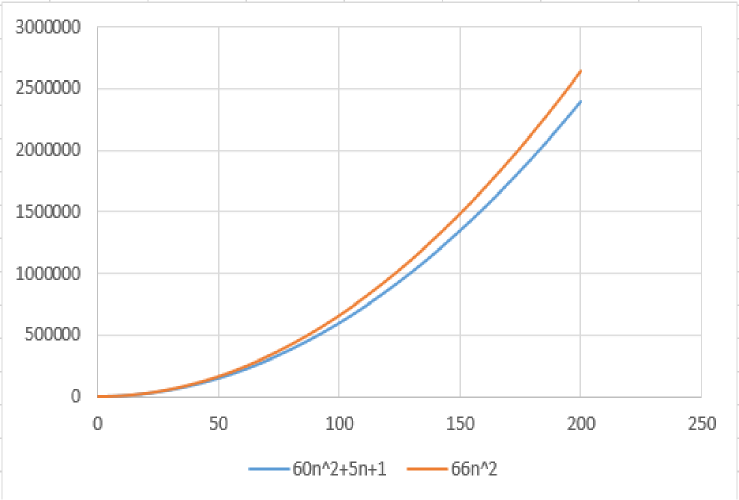
We need to find n_0 and c such that

$f(n) \geq c_1 g(n)$ and $f(n) \leq c_2 g(n)$ for all $n \geq n_0$.

- Let $f(n) = 8n + 128$, and $g(n) = n$.
- First we prove that $f(n) \in O(g(n))$
- Then we prove that $f(n) \in \Omega(g(n))$

Big-Θ (Big Theta)

- Prove that $f(n) \in O(g(n))$
- $\rightarrow f(n) \leq cg(n)$
- $\rightarrow 60n^2 + 5n + 1 \leq cn^2$
- $\rightarrow 60n^2 - cn^2 + 5n + 1 \leq 0$
- $\rightarrow (60 - c)n^2 + 5n + 1 \leq 0$

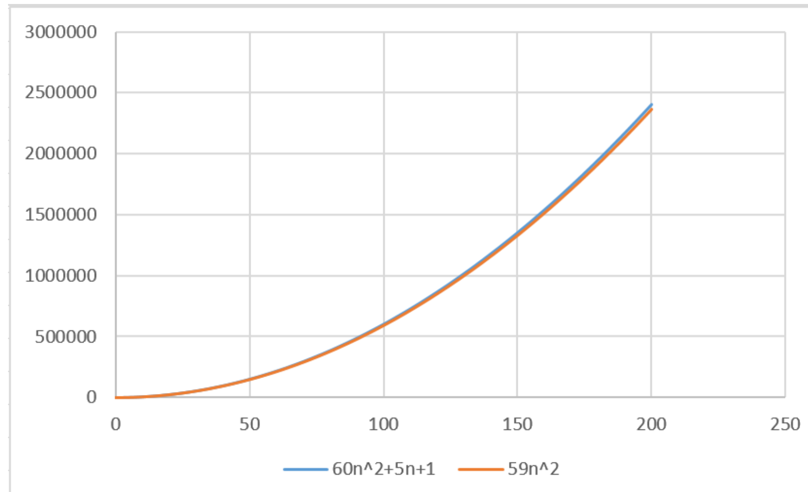


Pick $n_0 = 1$ and $c = 66$ and the inequality stands for all values of $n \geq n_0$ and $c = 66$. Hence, the statement $60n^2 + 50n + 1 \in \Theta(n^2)$ is true. The proof completes.

Big-Θ (Big Theta)

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

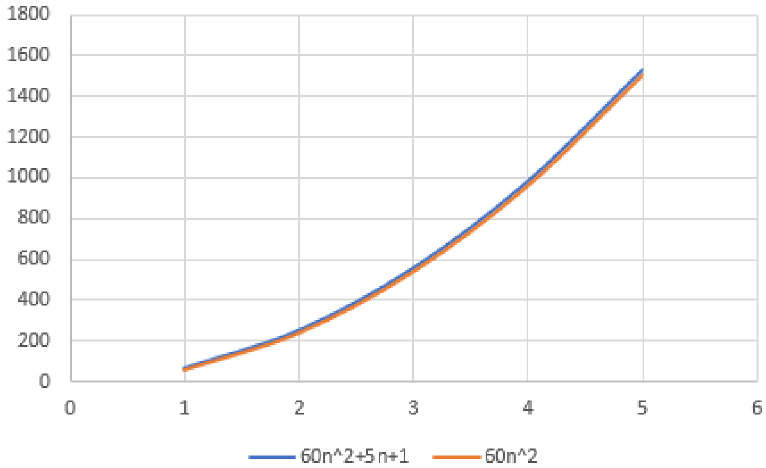
- Prove that $f(n) \in \Omega(g(n))$
- $\leftrightarrow f(n) \geq cg(n)$
- $\leftrightarrow 60n^2 + 5n + 1 \geq cn^2$
- $\leftrightarrow 60n^2 - cn^2 + 5n + 1 \geq 0$
- $\rightarrow (60 - c)n^2 + 5n + 1 \geq 0$



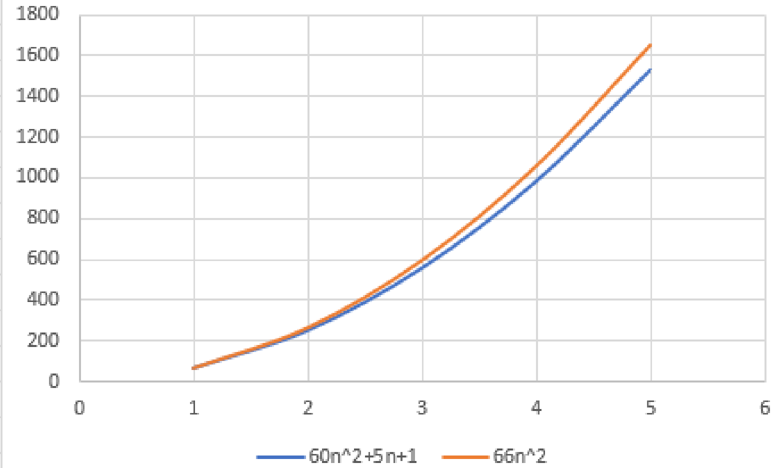
Pick $n_0 = 1$ and $c = 59$ and the inequality stands for all values of $n > n_0$ and $c = 59$. Hence, the statement $60n^2 + 50n + 1 \in \Theta(n^2)$ is valid. The proof completes.

Big-Θ (Big Theta)

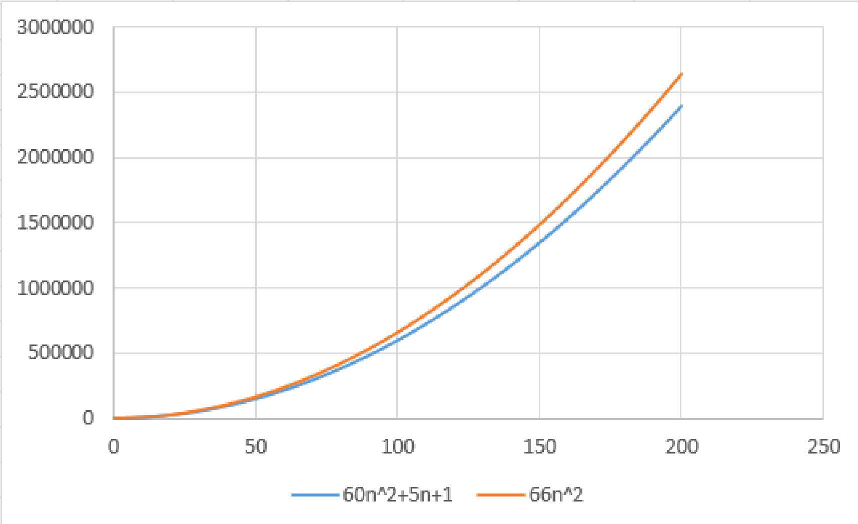
Hence, $f(n) \in \Omega(g(n))$ for some constant $c = 60$ and $n_0 = 1$; and $f(n) \in O(g(n))$ for some constant $c = 66$ and $n_0 = 1$.

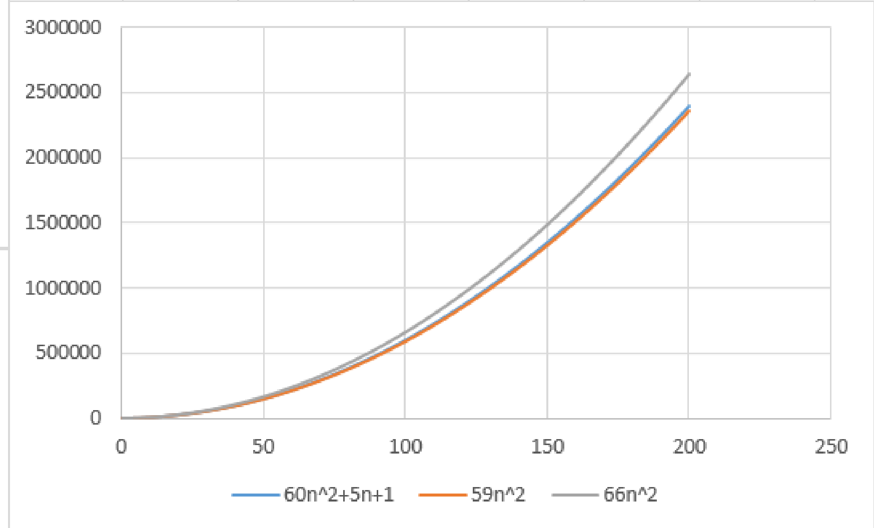


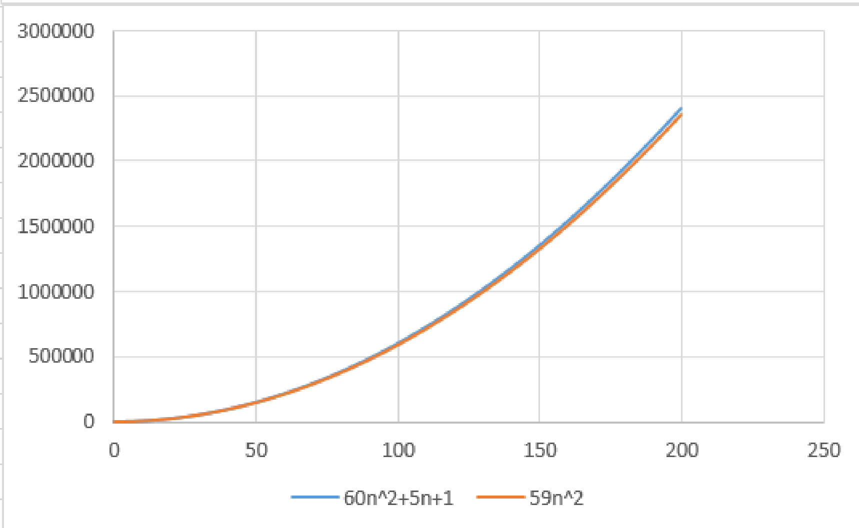
60n^2+5n+1 >= 60n^2



60n^2+5n+1 <= 66n^2


3000000
2500000
2000000
1500000
1000000
500000
0
0
50
100
150
200
250
60n^2+5n+1
66n^2


3000000
2500000
2000000
1500000
1000000
500000
0
0
50
100
150
200
250
60n^2+5n+1
59n^2
66n^2


3000000
2500000
2000000
1500000
1000000
500000
0
0
50
100
150
200
250
60n^2+5n+1
59n^2

Big-Θ (Big Theta)

Example: $2n + 3\lg n \in \Theta(n)$

Proof:

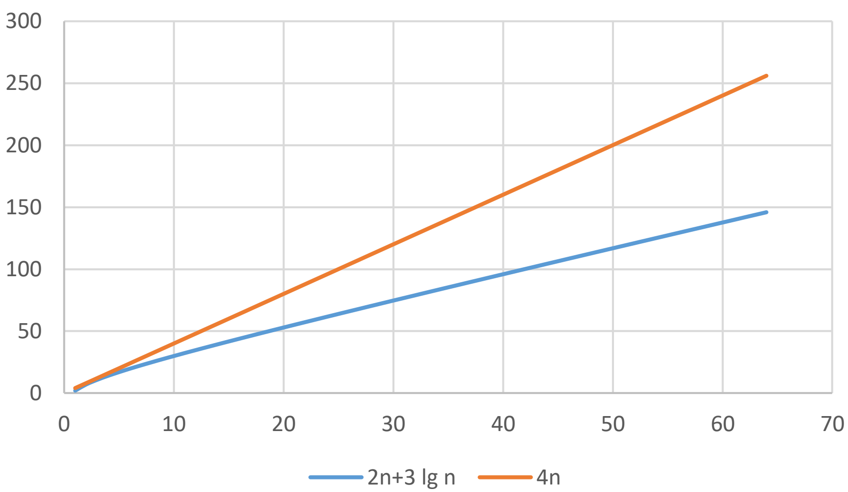
We need to find n_0 and c such that

$f(n) \geq c_1 g(n)$ and $f(n) \leq c_2 g(n)$ for all $n \geq n_0$.

- Let $f(n) = 2n + 3\lg n$, and $g(n) = n$.
- First we prove that $f(n) \in O(g(n))$
- Then we prove that $f(n) \in \Omega(g(n))$

Big-Θ (Big Theta)

- First prove that $f(n) \in O(g(n))$
- $\rightarrow f(n) \leq cg(n)$
- $\rightarrow 2n + 3\lg n \leq cn$
- $\rightarrow 2n - cn + 3\lg n \leq 0$
- $\rightarrow (2-c)n + 3\lg n \leq 0$

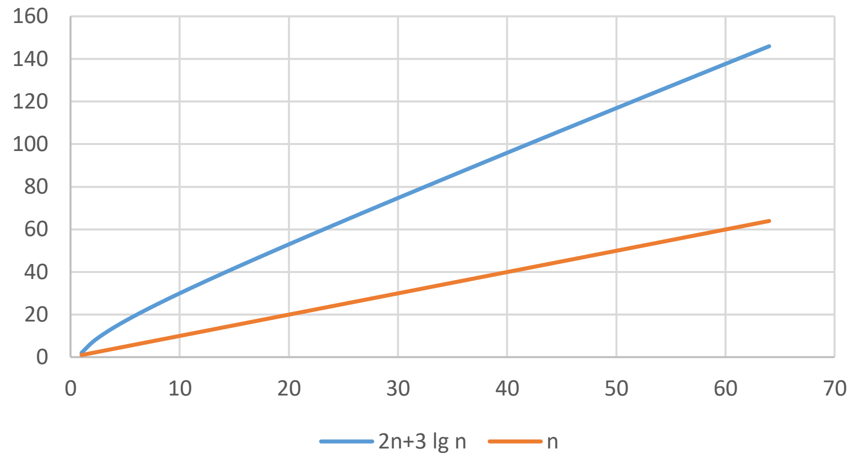


Note: In this module, I will use $\log_2 n$ or $\lg n$ to mean logarithm to the base of 2. If you see $\log n$, look at the context to interpret the base.

Pick $n_0 = 1$ and $c = 4$ and the inequality stands for any value of $n > n_0$ and $c = 4$. Hence the statement $f(n) \in O(g(n))$ is valid. The proof completes.

Big-Θ (Big Theta)

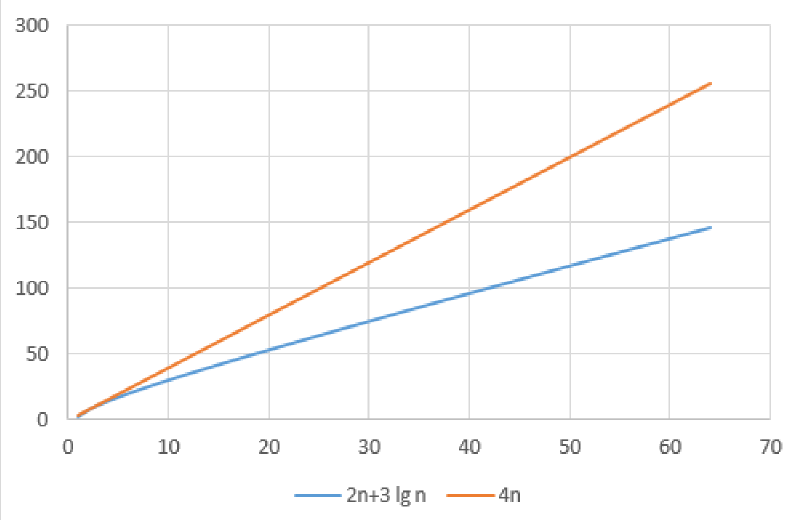
- Then prove that $f(n) \in \Omega(g(n))$
- $\rightarrow f(n) \geq cg(n)$
- $\rightarrow 2n + 3\lg n \geq cn$
- $\rightarrow 2n - cn + 3\lg n \geq 0$
- $\rightarrow (2 - c)n + 3\lg n \geq 0$

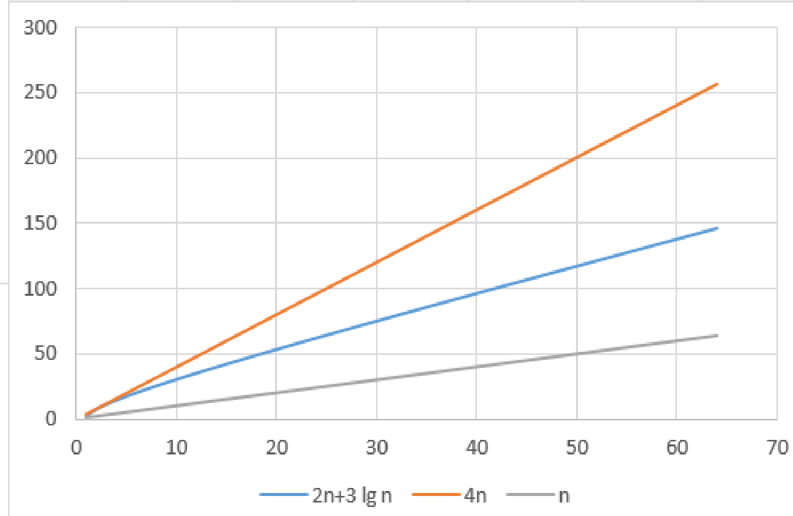


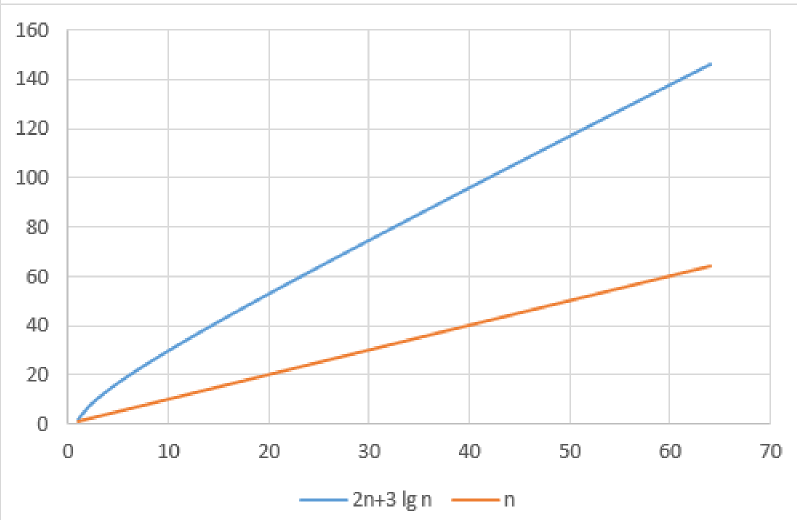
Pick $n_0 = 1$ and $c = 1$ and the inequality stands for all values of $n \geq n_0$. Hence the statement $f(n) \in \Omega(g(n))$ is valid. The proof completes.

Big-Θ (Big Theta)

- Hence, $f(n) \in \Omega(g(n))$ for some constant $c = 1$ and $n_0 = 1$; and $f(n) \in O(g(n))$ for some constant $c = 4$ and $n_0 = 1$.

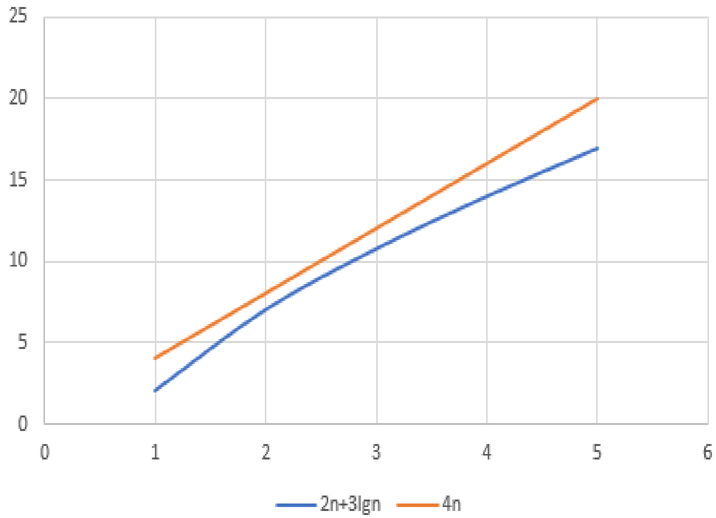

300
250
200
150
100
50
0
0 10 20 30 40 50 60 70
2n+3 lg n
4n


300
250
200
150
100
50
0
0 10 20 30 40 50 60 70
2n+3 lg n
4n
n


160
140
120
100
80
60
40
20
0
0 10 20 30 40 50 60 70
2n+3 lg n
n

Big-Θ (Big Theta)

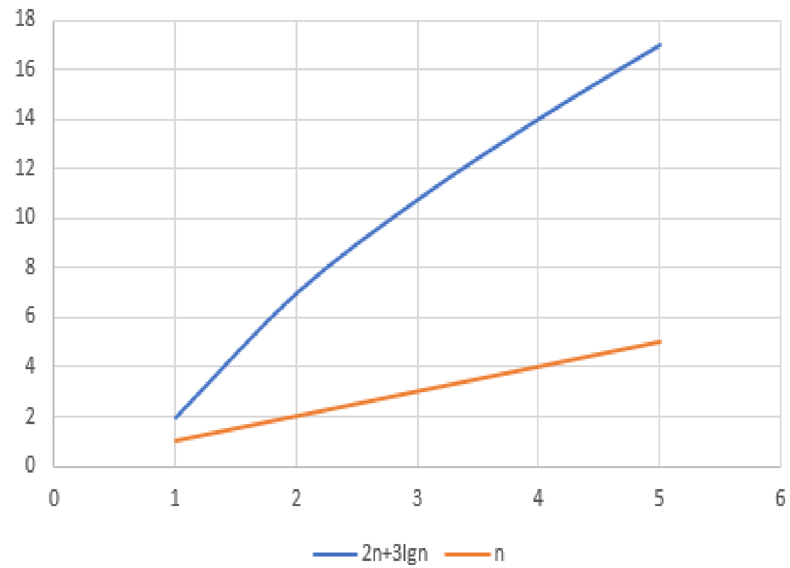
- Prove that $f(n) \in O(g(n))$
- $\leftrightarrow f(n) \leq cg(n)$
- $\leftrightarrow 2n + 3\lg n \leq cn$
- $\leftrightarrow 2n - cn + 3\lg n \leq 0$



Pick $n_0 = 1$ and $c = 4$ and the inequality stands.
Hence, the proof completes.

Big-Θ (Big Theta)

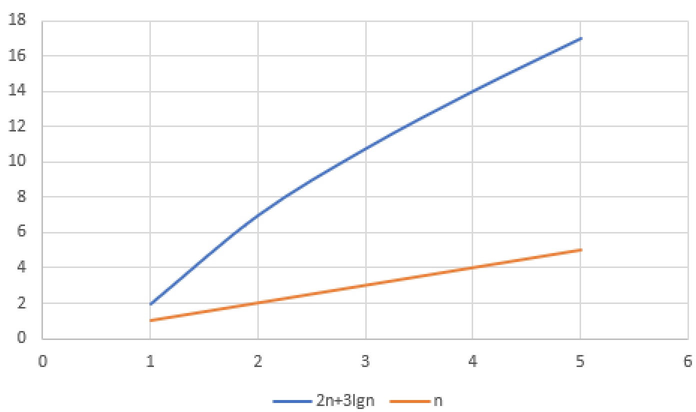
- Prove that $f(n) \in \Omega(g(n))$
- $\leftrightarrow f(n) \geq cg(n)$
- $\leftrightarrow 2n + 3\lg n \geq cn$
- $\leftrightarrow 2n - cn + 3\lg n \geq 0$



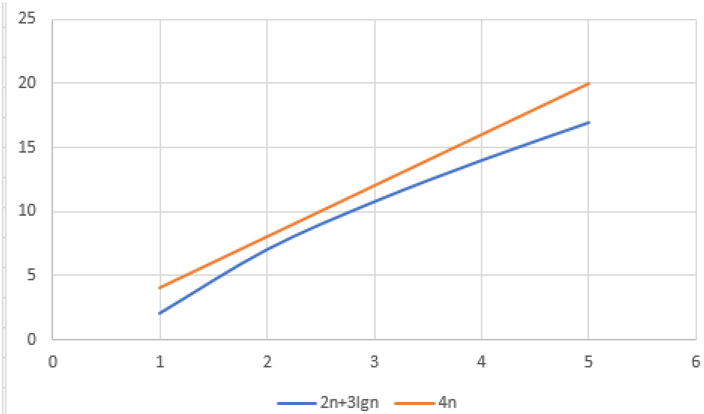
Pick $n_0 = 1$ and $c = 1$ and the inequality stands. Hence, the proof completes.

Big-Θ (Big Theta)

- Hence, $f(n) \in \Omega(g(n))$ for some constant $c = 1$ and $n_0 = 1$; and $f(n) \in O(g(n))$ for some constant $c = 4$ and $n_0 = 1$.



2n+3lgn >= 1n



2n+3lgn <= 4n

Properties of Asymptotic

- Suppose we know that $f_1(n) \in O\big(g_1(n)\big)$ and $f_2(n) \in O\big(g_2(n)\big)$, what can we say about the asymptotic behaviour of the sum of $f_1(n)$ and $f_2(n)$?

Properties of Asymptotic

Example:

Consider the functions $f_1(n) = n^3 + n^2 + n + 1 \in O(n^3)$ and $f_2(n) = n^2 + n + 1 \in O(n^2)$.

$$
\begin{array}{l}
f_1(n) = n^3 + n^2 + n + 1 \\
f_2(n) = \qquad\;\; n^2 + n + 1 \\
\hline
f_1(n) + f_2(n) = n^3 + 2n^2 + 2n + 2
\end{array}
$$

Hence, the asymptotic behaviour of the sum $f_1(n) + f_2(n)$ is $O(n^3)$

Properties of Asymptotic

- Suppose we know that $f_1(n) \in O\big(g_1(n)\big)$ and $f_2(n) \in O\big(g_2(n)\big)$. What can we say about the asymptotic behaviour of thee product of $f_1(n)$ and $f_2(n)$?

Example:

Consider the functions $f_1(n) = n^3 + n^2 + n + 1 \in O(n^3)$ and $f_2(n) = n^2 + n + 1 \in O(n^2)$.

The asymptotic behaviour of the product $f_1(n) \times f_2(n)$ is $O(n^3 \times n^2) = O(n^5)$.

Properties of Asymptotic

As an exercise, what is the asymptotic behaviour of the division $\frac{f_1(n)}{f_2(n)}$?

General plan for algorithm analysis

- Decide on parameter *n indicating input size*
- Identify algorithm's *basic operation*
- Determine *worst case for input of size n*
- May also need to determine the *average and best cases*
- Set up a sum expressing the number of times the algorithm's basic operation is executed
- Simplify the sum using standard formulas and rules to determine big-Oh of the algorithm's running time

Asymptotic rules

- If $f(n)$ is a polynomial of degree d, then $f(n)$ is $O\left(n^d\right)$, that is,
  - Drop lower-order terms
  - Drop constant factors
- Use the smallest possible class of functions
  - Say "$100n$ is $O(n)$" instead of "$100n$ is $O(n^2)$"
- Use the simplest expression of the class
  - Say "$\mathbf{3n+5 \text{ is } O(n)}$" **instead of** "$\mathbf{3n+5 \text{ is } O(3n)}$"