SINGAPORE INSTITUTE OF TECHNOLOGY

INF1001: Introduction to Computing

Part B

L3: Algorithms

Aug 2023

Outline

- Definition

- Algorithm Representation
  - Natural language
  - High level programming language
  - Flowchart
  - Pseudocode

- Algorithm Analysis (AA)
  - Time and space efficiency
  - Order of Magnitude (Big Oh)

Algorithm

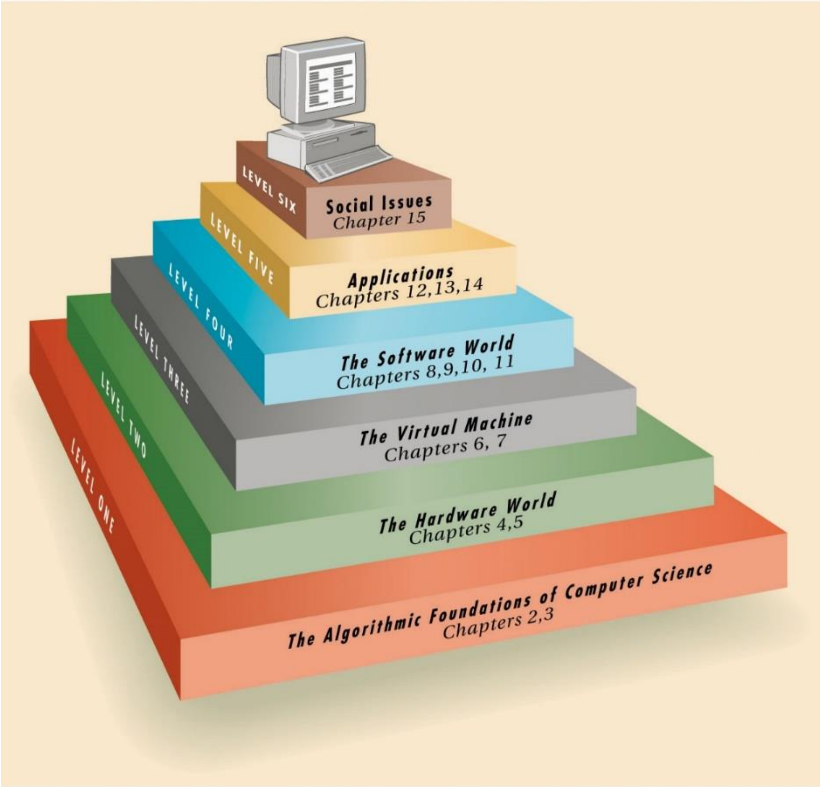
- Finite sequence of **rules/instructions** for carrying out some calculation or procedure

Find the sum of the first 100 numbers

i.e. what is 1 + 2 + 3 + . . + 100?

- Properties:
  - Correct
  - Terminate
  - Efficient: time, space (i.e. speed of execution, size of code)
  - Ease of understanding (not the most important, sometimes you don't want other to understand for security reasons)
  - Elegance

Why Study Algorithms?



- Impact is broad and far-reaching:
  - Internet, biology, computer, graphics, security, multimedia, social networks, physics

Algorithms

- The central concept underlying all computation
  - An **algorithm** is a step-by-step sequence of instructions for carrying out a task
- Programming: process of **designing and implementing algorithms** that a computer can carry out
- A programmer's job is to:
  - Create an algorithm for accomplishing a given objective, then
  - Translate the individual steps of the algorithm into a programming language that the computer can understand
- The use of algorithms is not limited to the domain of computing
  - e.g., recipes for baking cookies,

directions to your house

Terminology

Algorithm	• A set of steps that defines how a task is performed
Program	• A representation of an algorithm
Programming	• The process of developing a program
Software	• Programs and algorithms
Hardware	• Equipment

Algorithm representations

1. Natural language

2. High level programming language

3. Flowchart

4. Pseudocode

Algorithm representations:
1. Natural language

Algorithm for Adding two m-digit numbers

Initially, set the value of the variable *carry* to 0 and the value of the variable *i* to 0. When these initializations have been completed, begin looping as long as the value of the variable *i* is less than or equal to $(m - 1)$. First, add together the values of the two digits a_i and b_i and the current value of the carry digit to get the result called c_i. Now check the value of c_i to see whether it is greater than or equal to 10. If c_i is greater than or equal to 10, then reset the value of *carry* to 1 and reduce the value of c_i by 10; otherwise, set the value of *carry* to zero. When you are done with that operation, add 1 to *i* and begin the loop all over again. When the loop has completed execution, set the leftmost digit of the result c_m to the value of *carry* and print out the final result, which consists of the digits $c_m\ c_m - 1 \ldots c_0$. After printing the result, the algorithm is finished, and it terminates.

- Language spoken and written in everyday life

<u>Problems with Natural Language:</u>

- Can be extremely **verbose** – rambling, unstructured, hard to follow
- Too "**rich**" in interpretation and meaning

Algorithm representations:

2. High Level Programming Language

Algorithm for Adding two m-digit numbers

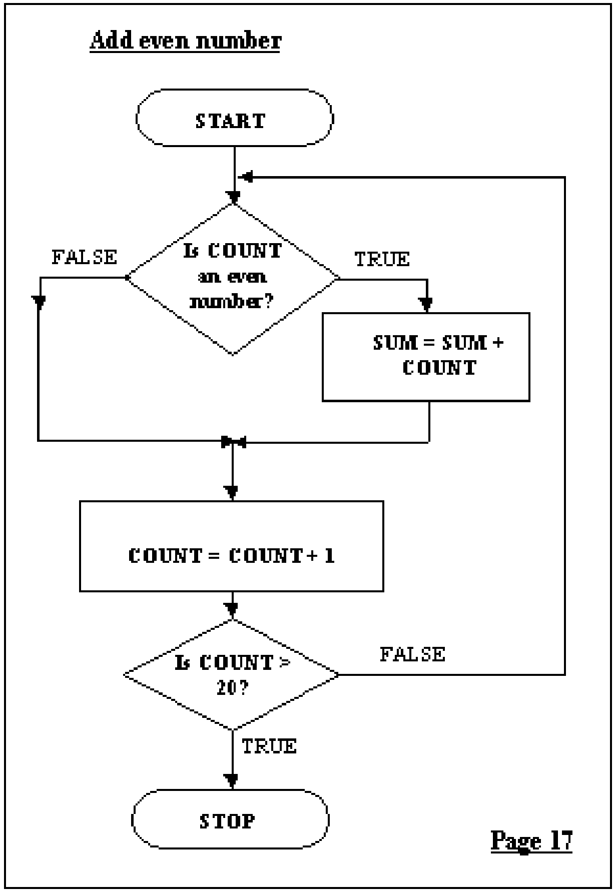
```
{
int i, m, Carry;
int[] a = new int[100];
int[] b = new int[100];
int[] c = new int[100];
m = Console.readInt();
for (int j = 0; j < =m−1; j++)    {
            a[j] = Console.readInt();
            b[j] = Console.readInt();
}
Carry = 0;
i = 0;
while (i < m)    {
            c[i] = a[i] + b[i] + Carry;
            if (c[i] > =10)
            .
            .
            .
```

Actual programming language used in computers. Examples: C++, Java, etc.

Problems with using a high-level programming language for algorithms

- Can be difficult – chances you will make **mistakes** with trying to understand the logic and making sure that there are no syntactic and semantics errors.
- During the initial phases of design, forced to deal with **detailed** language issues.
- Too **specific** – what if you want to write in another language instead?

Algorithm representations:
3. Flowchart



Symbol	Name	Function
	Start/end	An oval represents a start or end point
	Arrows	A line is a connector that shows relationships between the representative shapes
	Input/Output	A parallelogram represents input or output
	Process	A rectangle represents a process
	Decision	A diamond indicates a decision

Extracted from https://www.smartdraw.com/flowchart/flowchart-symbols.htm

Algorithm representations:
4. Pseudocode

- English language constructs modeled to look like statements available in most programming languages
- Steps presented in a structured manner (numbered, indented, and so on)
- No fixed syntax for most operations is required
- Emphasis is on process, not notation
- Well-understood forms allow logical reasoning about algorithm behavior

Algorithm representations:

4. Pseudocode

SINGAPORE INSTITUTE OF TECHNOLOGY

Algorithm for Adding two m-digit numbers

Initially, set the value of the variable *carry* to 0 and the value of the variable *i* to 0. When these initializations have been completed, begin looping as long as the value of the variable *i* is less than or equal to ($m - 1$). First, add together the values of the two digits a_i and b_i and the current value of the carry digit to get the result called c_i.

ether it is greater than or equal to 10. If c_i is

et the value of *carry* to 1 and reduce the value of

carry to zero. When you are done with that oper-

ll over again. When the loop has completed exe-

sult c_m to the value of *carry* and print out the

its c_m $c_m - 1 \dots c_0$. After printing the result, the

es.

Algorithm:

Step 1 Set the value of *carry* to 0.

Step 2 Set the value of *i* to 0.

Step 3 While the value of *i* is less than or equal to $m - 1$, repeat the instructions in steps 4 through 6.

Step 4 Add the two digits a_i and b_i to the current value of *carry* to get c_i.

Step 5 If $c_i \geq 10$, then reset c_i to $(c_i - 10)$ and reset the value of *carry* to 1; otherwise, set the new value of *carry* to 0.

Step 6 Add 1 to *i*, effectively moving one column to the left.

Step 7 Set c_m to the value of *carry*.

Step 8 Print out the final answer, $c_m\ c_{m\text{-}1}\ c_{m\text{-}2} \dots c_0$.

Step 9 Stop.

Types of Algorithmic Operations

1. Sequential
   a. Executes its instructions in a straight line from top to bottom and then stops
   b. Computation, input and output operations (GET, SET)

2. Conditional
   a. If-else

3. Iterative
   a. For/while loops

Algorithmic Operations:
1. Sequential operations

- Perform a single task
  - **Input**: gets data values from outside the algorithm
  - **Computation**: a single numeric calculation
  - **Output**: sends data values to the outside world
- A **variable** is a named location to hold a value
- A **sequential algorithm** is made up only of sequential operations
- Example: algorithm to print the average miles per gallon, used by a car:

Step	Operation
1	Get values for *gallons used*, *starting mileage*, *ending mileage*
2	Set value of *distance driven* to (*ending mileage* – *starting mileage*)
3	Set value of *average miles per gallon* to (*distance driven* ÷ *gallons used*)
4	Print the value of *average miles per gallon*
5	Stop

Algorithmic Operations:
1. Sequential operations

- A purely sequential algorithm, such as the previous example, is sometimes also termed as **straight line algorithm**
- Most real world problems are not straight line
- To solve them, we need non-sequential operations such as **branching** and **repetition**
- **Conditional** operations mimic branching
- **Iterative** operations mimic repetition

Algorithmic Operations:

2. Conditional operations

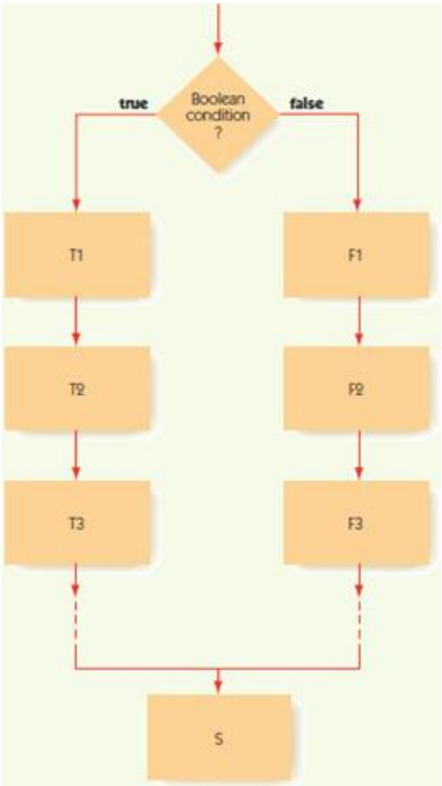
- Conditional operations <u>ask</u> questions and <u>select</u> the next operation on the basis of the answer
- Usually follow this format:

If "a true/false condition" is **true** then

First set of algorithmic operations is performed

Else (or otherwise)

Second alternative set of algorithmic operations is performed instead



Algorithmic Operations:
2. Conditional operations

- Extending the previous example to print the average miles per gallon, this time we also print whether the driver is getting good gas mileage or not:

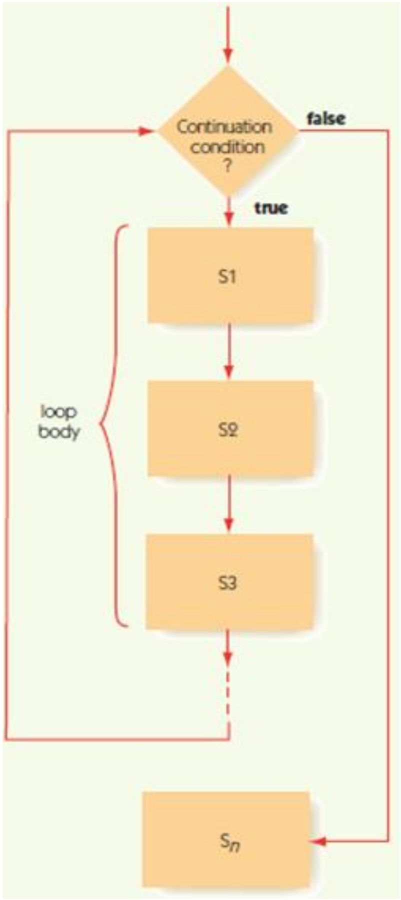
Step	Operation
1	Get values for *gallons used*, *starting mileage*, *ending mileage*
2	Set value of *distance driven* to (*ending mileage* – *starting mileage*)
3	Set value of *average miles per gallon* to (*distance driven* ÷ *gallons used*)
4	Print the value of *average miles per gallon*
5	If *average miles per gallon* is greater than 25.0 then
6	Print the message 'You are getting good gas mileage'
	Else
7	Print the message 'You are NOT getting good gas mileage'
8	Stop

Algorithmic Operations:
3. Iterative operations

- **Iterative** operations mimic repetition
- **Loop** refers to repetition of block of instructions
  - Often, the real power of a computer comes from performing a calculation many times
  - Looping takes advantage of the power of computers
- One format is the **while** statement:

```
While ("a true/false condition") do step i to step j
  Step i:          operation
  Step i+1:        operation
  .
  .
  Step j:          operation
```

- If the continuation condition never becomes false, then we will forever be trapped in an **infinite loop**. This is also known as the forever loop scenario that result in software to "hang" as they are stuck in loop that never terminates.



Algorithmic Operations:
3. Iterative operations

- Extending the previous example to print whether the driver is getting good gas mileage or not, this time we print only while the user respond with a Yes.

Step	Operation
1	*response* = *Yes*
2	While (*response* = *Yes*) do Steps 3 through 11
3	Get values for *gallons used*, *starting mileage*, *ending mileage*
4	Set value of *distance driven* to (*ending mileage* – *starting mileage*)
5	Set value of *average miles per gallon* to (*distance driven* ÷ *gallons used*)
6	Print the value of *average miles per gallon*
7	If average miles per gallon > 25.0 then
8	Print the message 'You are getting good gas mileage'
	Else
9	Print the message 'You are NOT getting good gas mileage'
10	Print the message 'Do you want to do this again? Enter Yes or No'
11	Get a new value for *response* from the user
12	Stop

Third version of the average miles per gallon algorithm

Designing and Analysing Algorithms

Four steps to solving problems (George Polya):

1. Understand the problem

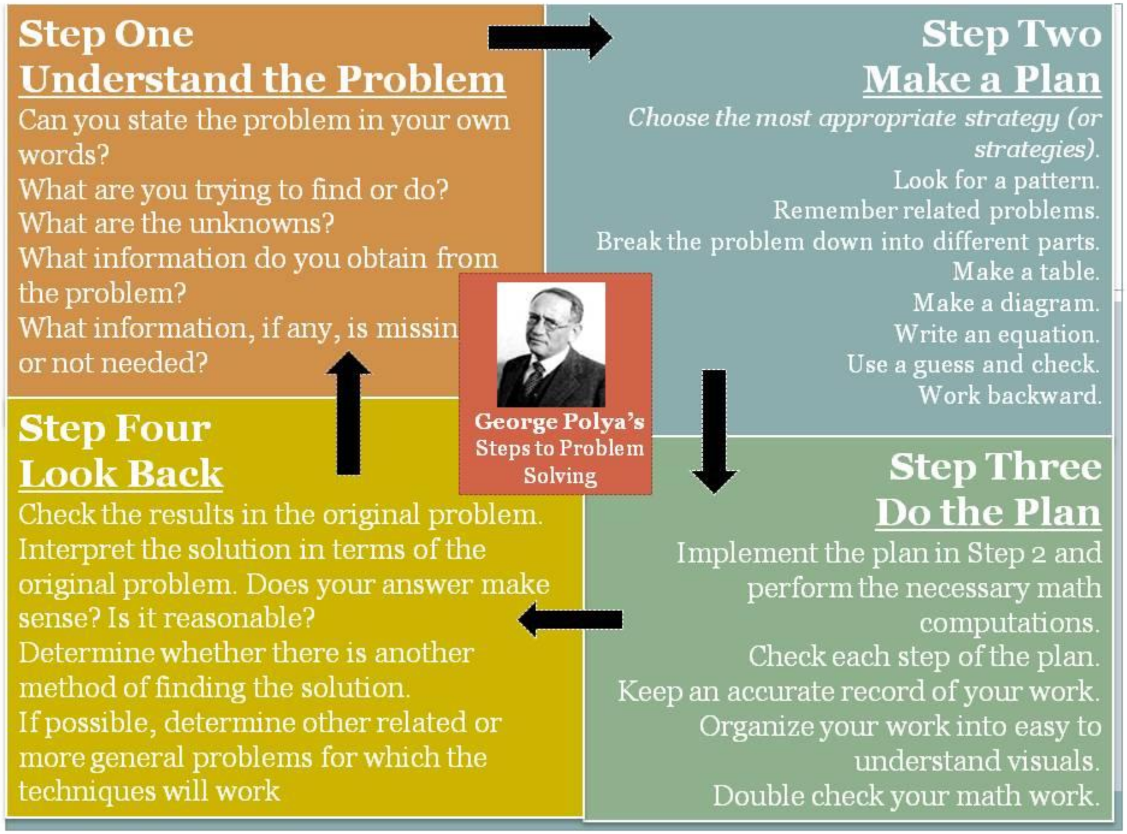
2. Devise a plan

3. Carry out your plan

4. Examine the solution

More details at:

https://math.berkeley.edu/~gmelvin/polya.pdf



Algorithm Analysis (AA)

- AA is the study of the **efficiency** of various algorithms
- Determine the amount of resources (time and storage) necessary to execute it
  - Running time of an algorithm typically grows with the input size
- Why do we do AA?

1. Classify problems by difficulty
2. Predict performance, compare algorithms, tune parameters
3. Better understand and improve implementations and algorithms
4. Intellectual challenge:
   - Algorithm analysis is more interesting than programming

Algorithm Analysis (AA)

- There are two ways to analyse an algorithm:
  - 1. Empirical studies
  - 2. Theoretical analyses

- **1. Empirical studies**
  - Results based on direct observation or experiments
  - Write a program and run it with varying input size
  - Get accurate measurement of actual running time

- **2. Theoretical analyses**
  - Mathematical model
  - E.g. Total running time = Sum of cost × frequency of execution of all program statements

Algorithm Analysis (AA)

- Sometimes computer programs look very similar
  - Which program is better?

- There is an important difference between a **program** and the underlying **algorithm** that the program is representing
  - **Algorithm** is a generic, step-by-step list of instructions for solving problem
  - **Program** is an algorithm that has been encoded into some programming language. *There may be <u>many</u> programs for <u>same</u> algorithm*

https://leetcode.com/problems/sudoku-solver/discuss/15796/Singapore-prime-minister-Lee-Hsien-Loong's-Sudoku-Solver-code-runs-in-1ms

http://vivian.balakrishnan.sg/sudoku/

Examples

When two programs solve the same problem but look different, which is better?

```python
def sumOfN(n):
   theSum = 0
   for i in range(1,n+1):
       theSum = theSum + i

   return theSum

print(sumOfN(10))
```

VS

```python
def foo(tom):
    fred = 0
    for bill in range(1,tom+1):
       barney = bill
       fred = fred + barney

    return fred

print(foo(10))
```

- Criteria:
  - Readability, OR
  - Efficiency, i.e., computing resources

  (space/memory & time/speed) used

Example: Algorithm 1

- Benchmark analysis for execution time

```
import time

def sumOfN2(n):
   start = time.time()

   theSum = 0
   for i in range(1,n+1):
      theSum = theSum + i

   end = time.time()

   return theSum,end-start
```

Note start and end time

```
>>>for i in range(5):
       print("Sum is %d required %10.7f seconds"%sumOfN(10000))
Sum is 50005000 required  0.0018950 seconds
Sum is 50005000 required  0.0018620 seconds
Sum is 50005000 required  0.0019171 seconds
Sum is 50005000 required  0.0019162 seconds
Sum is 50005000 required  0.0019360 seconds
```

Invoking 5 times each computing sum of first 10,000 integers

Avg approx. = 0.0019s

```
>>>for i in range(5):
       print("Sum is %d required %10.7f seconds"%sumOfN(100000))
Sum is 5000050000 required  0.0199420 seconds
Sum is 5000050000 required  0.0180972 seconds
Sum is 5000050000 required  0.0194821 seconds
Sum is 5000050000 required  0.0178988 seconds
Sum is 5000050000 required  0.0188949 seconds
>>>
```

Invoking 5 times each computing sum of first 100,000 integers

Avg approx. = 0.019s

```
>>>for i in range(5):
       print("Sum is %d required %10.7f seconds"%sumOfN(1000000))
Sum is 500000500000 required  0.1948988 seconds
Sum is 500000500000 required  0.1850290 seconds
Sum is 500000500000 required  0.1809771 seconds
Sum is 500000500000 required  0.1729250 seconds
Sum is 500000500000 required  0.1646299 seconds
>>>
```

Invoking 5 times each computing sum of first 1,000,000 integers

Ave approx. = 0.19s

Example: Algorithm 2

- Benchmark analysis for execution time

```python
def sumOfN3(n):
    return (n*(n+1))/2

print(sumOfN3(10))
```

```
Sum is 50005000 required 0.00000095 seconds
Sum is 5000050000 required 0.00000191 seconds
Sum is 500000500000 required 0.00000095 seconds
Sum is 50000005000000 required 0.00000095 seconds
Sum is 5000000050000000 required 0.00000119 seconds
```

- Looking at the 2 different algorithms, we observe:
  - iterative solutions do more work
  - time required increases as input size increase
- But, do you get the same result if you run on different computer?

Empirical Studies – Problem

- System dependent:
  - Hardware, software, system
- Difficult to get precise measurements
- A different way to analyse:
  - independent of program and computer use
  - theoretical analysis and using mathematical models

Mathematical Models

- Want to characterize algorithm efficiency in terms of execution time, ***independent*** of any program or computer
  - Quantify number of operations that the algorithm will require
- Basic unit of computation: **Count number of assignment statements**

```python
def sumOfN(n):
    theSum = 0
    for i in range(1,n+1):
        theSum = theSum + i

    return theSum

print(sumOfN(10))
```

```
1 assignment to theSum
1 assignment to i
1 assignment to theSum
```

```
Total number of assignments  = 1+2n
```

Goal: show how the algorithm's execution time changes with respect to the size of the problem.

Mathematical Models

- Denote total 1+2n as a special function: T(n) where n is the size of the problem.
- T(n) is the time it takes to solve a problem of size n, in our previous example T(n) = 1+2n
- The exact number of operations is not as important as determining the **most dominant** part of T(n)
  - As n gets really big, the first constant term (1) is insignificant to the second term 2n
  - As problem gets larger, some portion of T(n) overpowers the rest
  - E.g. for T(n) = 2n + 1:
    - When n is small, e.g. if n = 1, the influence over T(n) is significant since T(n) = 2×1 + 1 = 3
    - But if n very large, e.g. if n = 1,000,000, the influence over T(n) is now negligible since

      T(n) = 2,000,000 + 1 = 2,000,001

      T(n) ≈ 2,000,000

We are interested in how the algorithm scales

Big-O Notation

- Interested in the most dominant part of T(n) which is used for comparison
- The **order of magnitude function** describes part of T(n) that increases the fastest as value n increases
  - Provides useful approximation to the actual number of steps in the function
  - In our previous example T(n) = 1+2n, we say the running time is O(n)

- For input size, must consider behavior for:
  - Best case
  - Worst case
  - Average case

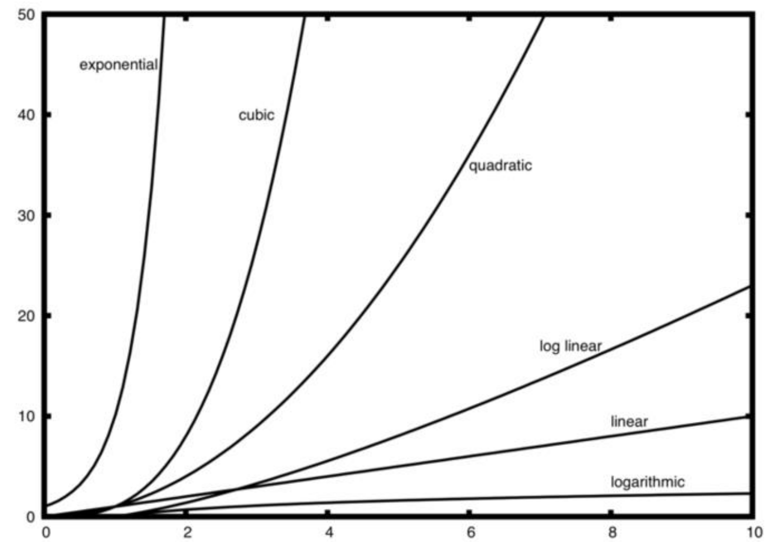
Common Big O Notations

O	Name
O(1)	Constant
O(log n)	Logarithmic
O(n)	Linear
O(n log n)	Log Linear
$O(n^2)$	Quadratic
$O(n^3)$	Cubic
$O(2^n)$	Exponential
O(n!)	Factorial

Best

Worst

Running Time Complexity

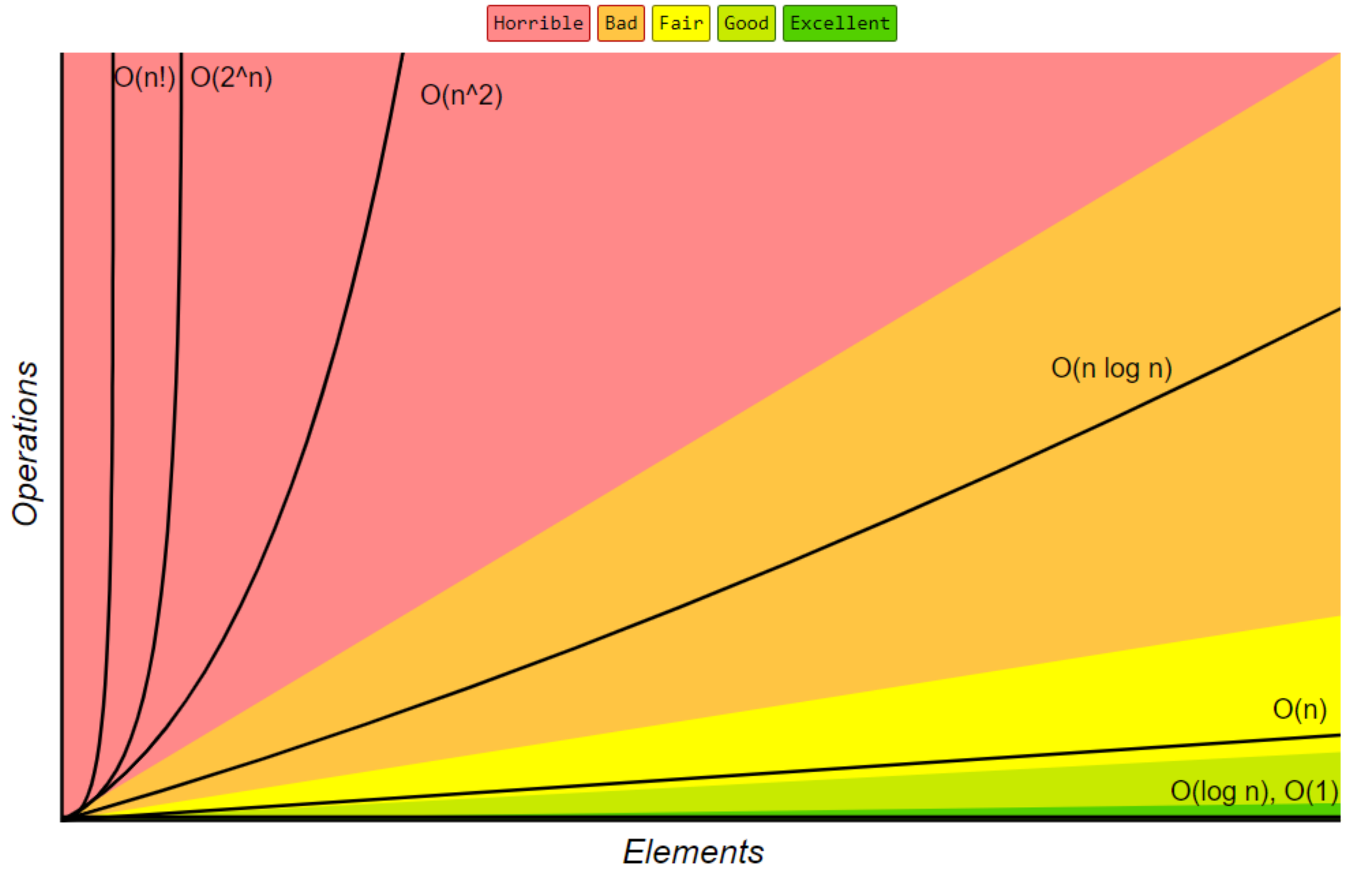


Input size (n)

- Whatsapp has about 65B messages a day, or ~750,000 messages per second!
- Alibaba Taobao has on average 9 million transactions per day or ~100 per second!

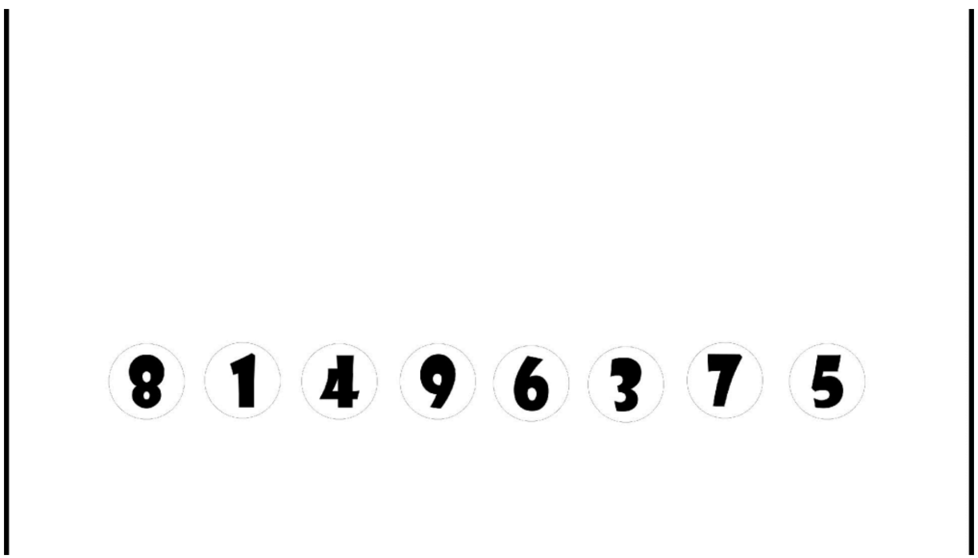
ORDER	n = 10	50	100	1,000
lg n	0.0003 sec	0.0006 sec	0.0007 sec	0.001 sec
n	0.001 sec	0.005 sec	0.01 sec	0.1 sec
n^2	0.01 sec	0.25 sec	1 sec	1.67 min
2^n	0.1024 sec	3,570 years	4 x 10^{16} centuries	*Too big to compute!!*

Big-O complexity chart



Example 1: Sequential Search

- Search for number X among a (unsorted) list of *n* number
- Start at the beginning and compare X to each entry until a match is found




```
i = 1
found = false
while i <= n && found==false do
begin
  if X == a[i] then
    found = true
  else
    i = i+1
end
if found==true then
  report "Found!"
else report "Not Found!"
```

See https://www.youtube.com/watch?v=x1d1b6Rb--E

- Sequential Search: 0:00 to 0:29
- Binary Search: 0:30 to 1:24

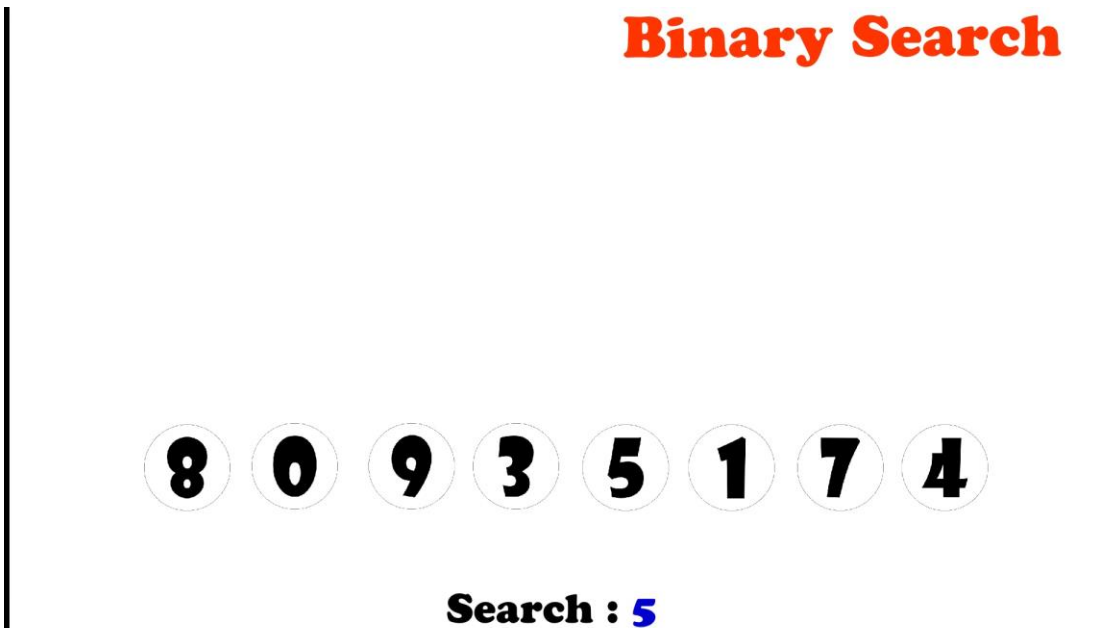
Example 1: Sequential Search

Best case	Average case	Worst case
X is the first value on the list	X is in the middle of the list	X is the last number in the list X is not in the list
1 comparison	Roughly n/2 comparisons	n comparisons
O(1)	O(n)	O(n)

Space efficiency

- Memory to store the list plus the value that is being searched
- Very space efficient

Example 2: Binary Search



Needs the list to be already sorted

```
BinarySearch(list[], min, max, key)
while min ≤ max do
   mid = (max+min) / 2
   if list[mid] >key then
      max = mid-1
   else if list[mid] <key then
      min = mid+1
   else
      return mid
   end if
end while
return false
```

See https://www.youtube.com/watch?v=x1d1b6Rb--E

- Sequential Search: 0:00 to 0:29
- Binary Search: 0:30 to 1:24

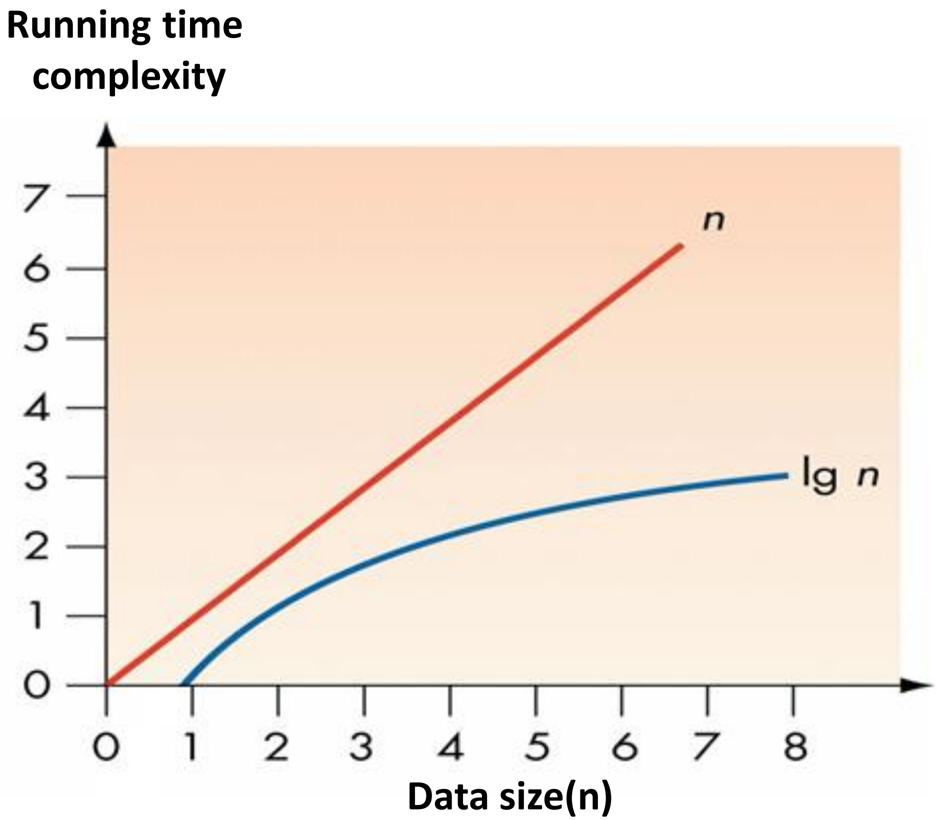
Example 2: Binary Search

- **More efficient** than sequential search in Example 1
- But works ONLY if search list is already **sorted/ordered**
  - Search for value by comparing to middle element
  - If not a match, restrict search to either lower or upper half only
  - ***Each pass eliminates half the data***

Best case	Average case	Worst case
Value is in the middle	Value is somewhere in the list	Value not in the list
1 comparison	$\log_2 n$ comparisons	$\log_2 n$ comparisons $\log_2 n$: number of times n can be divided by two before reaching 1
O(1)	O(log n)	$O(\log_2 n)$ or also known simply as O(log n)

O(n) versus O(log n)

- Tradeoffs:
  - Sequential search - O(n)
    - Slower, but works on unordered data
  - Binary search – O(log n)
    - Faster (much faster), but data must be sorted first



Example 3: Selection Sort

```
for (j = 0; j < n-1; j++)
    int iMin = j;
    for (i = j+1; i < n; i++)
      if (a[i] < a[iMin])
        iMin = i;
    if (iMin != j)
      swap(a[j], a[iMin]);
```

What is the amount of work done?

See https://www.youtube.com/watch?v=g-PGLbMth_g

Example 3: Selection Sort

TWO (2) types of work done here: **comparison** and **exchanges**

- For *comparison*:
  - Given n values, does (n-1) comparison
  - Total comparison cost = (n-1) + (n-2) + … + 2 + 1 = n(n-1)/2

$$\left(\frac{n-1}{2}\right)n = \frac{1}{2}n^2 - \frac{1}{2}n$$

  - At large n, it is taken as **O(n^2)**

```
for (j = 0; j < n-1; j++)
   int iMin = j;
   for (i = j+1; i < n; i++)
     if (a[i] < a[iMin])
        iMin = i;
   if (iMin != j)
     swap(a[j], a[iMin]);
```

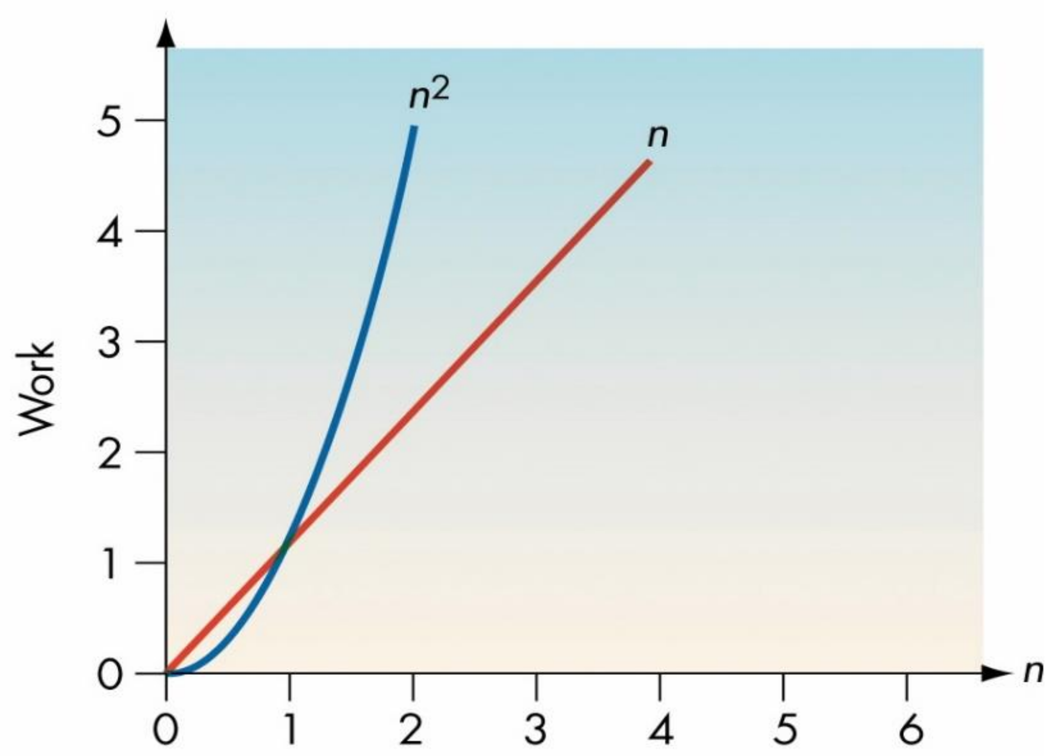
- For *exchanges*:
  - Given n values, does n (swapping largest into place) exchanges so O(n)

- Total : O(n^2)+ O(n)
  - At large n, exchanges O(n) are negligible compared to the work done for comparisons, **O(n^2)**
  - Therefore, ignoring the work done by exchanges, the total work done is taken as **O(n^2)**

O(n) vs O(n²)

- Anything that is $\Theta(n^2)$ will eventually have larger values than anything that is $\Theta(n)$, no matter what the constants

- An algorithm that runs in time $\Theta(n)$ will <u>outperform</u> one that runs in $\Theta(n^2)$



Example 3: Selection Sort

Therefore, the efficiency is seen as follows:

Best case	Average case	Worst case
$O(n^2)$	$O(n^2)$	$O(n^2)$

```
for (j = 0; j < n-1; j++)
    int iMin = j;
    for (i = j+1; i < n; i++)
      if (a[i] < a[iMin])
        iMin = i;
    if (iMin != j)
      swap(a[j], a[iMin]);
```

Space efficiency

- Space for the input sequence, plus a constant number of local variables

If you need more thorough explanation, please read pg. **89 to 93 of Chapter 3**, The Efficiency of Algorithms, Invitation to Computer Science, 5th Edition

Summary Big Oh Notation

O(1):

- Usually does not contain loop, recursion or call to any other non-constant function
- A loop that runs **constant** number of times with **no n**

```
// Here c is a constant
for (int i = 1; i <= c; i++) {
    // some O(1) expressions
}
```

Summary Big Oh Notation

O(n)

- Loop executes **n times** that **increment/decrement** by constant amount

```
n = 16;
m = 4;

for (i=1; i<=n; i++) {
  printf("Loop i = %d\n", i);
}

printf("\n\n");

for (i=n; i>=1; i--) {
  printf("Loop i = %d\n", i);
}
```

```
Loop i = 1
Loop i = 2
Loop i = 3
Loop i = 4
Loop i = 5
Loop i = 6
Loop i = 7
Loop i = 8
Loop i = 9
Loop i = 10
Loop i = 11
Loop i = 12
Loop i = 13
Loop i = 14
Loop i = 15
Loop i = 16
```

```
Loop i = 16
Loop i = 15
Loop i = 14
Loop i = 13
Loop i = 12
Loop i = 11
Loop i = 10
Loop i = 9
Loop i = 8
Loop i = 7
Loop i = 6
Loop i = 5
Loop i = 4
Loop i = 3
Loop i = 2
Loop i = 1
```

Summary Big Oh Notation

O(n²)

- Time complexity of nested loop equal to **number of times innermost statement is executed**

```c
n = 16;
m = 4;

printf("Demonstrating O(n^2) where m=4\n");
for (i=1; i<=m; i++) {
  for (j=1; j<=m; j++) {
    printf(">>>>Inner Loop j = %d\n", j);
  }
  printf("Outer Loop i = %d\n", i);
}

printf("\n\n");

printf("Demonstrating O(n^2) where m=4\n");
for (i=m; i>=1; i--) {
  for (j=1; j<=m; j++) {
    printf(">>>>Inner Loop j = %d\n", j);
  }
  printf("Outer Loop i = %d\n", i);
}
```

```
Demonstrating O(n^2) where m=4
>>>>Inner Loop j = 1
>>>>Inner Loop j = 2
>>>>Inner Loop j = 3
>>>>Inner Loop j = 4
Outer Loop i = 1
>>>>Inner Loop j = 1
>>>>Inner Loop j = 2
>>>>Inner Loop j = 3
>>>>Inner Loop j = 4
Outer Loop i = 2
>>>>Inner Loop j = 1
>>>>Inner Loop j = 2
>>>>Inner Loop j = 3
>>>>Inner Loop j = 4
Outer Loop i = 3
>>>>Inner Loop j = 1
>>>>Inner Loop j = 2
>>>>Inner Loop j = 3
>>>>Inner Loop j = 4
Outer Loop i = 4
```

```
Demonstrating O(n^2) where m=4
>>>>Inner Loop j = 1
>>>>Inner Loop j = 2
>>>>Inner Loop j = 3
>>>>Inner Loop j = 4
Outer Loop i = 4
>>>>Inner Loop j = 1
>>>>Inner Loop j = 2
>>>>Inner Loop j = 3
>>>>Inner Loop j = 4
Outer Loop i = 3
>>>>Inner Loop j = 1
>>>>Inner Loop j = 2
>>>>Inner Loop j = 3
>>>>Inner Loop j = 4
Outer Loop i = 2
>>>>Inner Loop j = 1
>>>>Inner Loop j = 2
>>>>Inner Loop j = 3
>>>>Inner Loop j = 4
Outer Loop i = 1
```

Summary Big Oh Notation

O(log n)

- Loop executes **n times** that is **divided/multiplied** by constant amount

```
n = 16;
m = 4;

for (i=1; i<=n; i*=2) {
    printf("Loop i = %d\n", i);
}

printf("\n\n");

for (i=n; i>0; i/=2) {
    printf("Loop i = %d\n", i);
}
```

```
Loop i = 1
Loop i = 2
Loop i = 4
Loop i = 8
Loop i = 16


Loop i = 16
Loop i = 8
Loop i = 4
Loop i = 2
Loop i = 1
```

Note that to calculate $\log_2 16$ using calculator, input this:

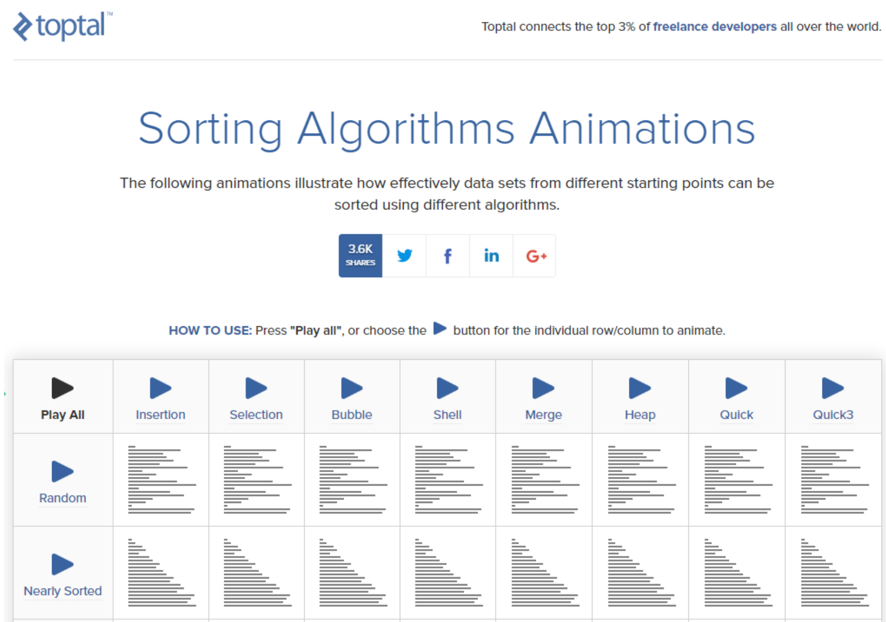
$\log_{10} 16 / \log_{10} 2$

Refer to https://www.calculator.net/log-calculator.html

Algorithms Visualization

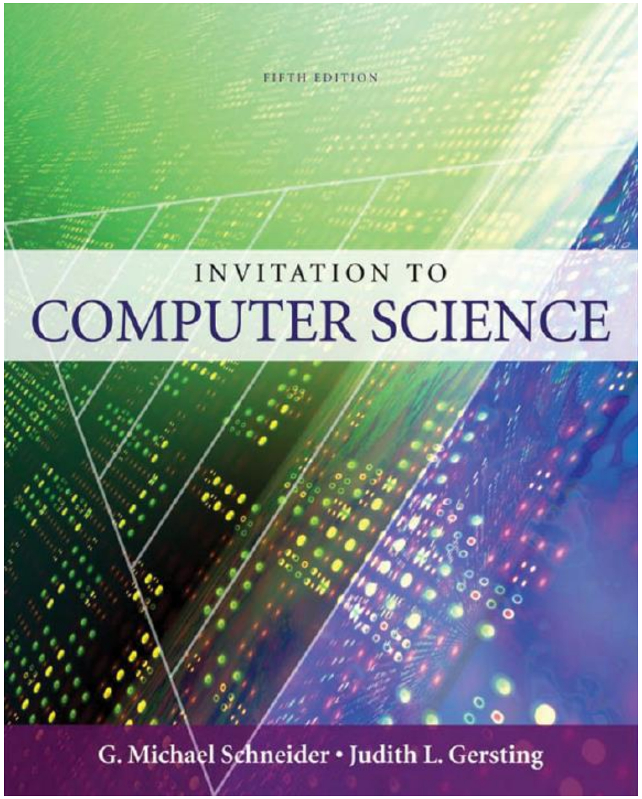
SINGAPORE INSTITUTE OF TECHNOLOGY

- http://www.sorting-algorithms.com/
- https://visualgo.net/en
- http://www.cs.usfca.edu/~galles/visualization/Algorithms.html



References

- *Chapter 2 Algorithm Discovery & Design;*
- *Chapter 3 The Efficiency of Algorithms*, An Invitation to Computer Science, 5th Edition, G. Michael Schneider, Judith L. Gersting, CENGAGE Learning



Summary

- ✓ Definition
- ✓ Algorithm Representation
  - Natural language
  - High level programming language
  - Flowchart
  - Pseudocode
- ✓ Algorithm Analysis (AA)
  - Time and space efficiency
  - Order of Magnitude (Big Oh)
- To be continued in your Data Structures and Algorithms module