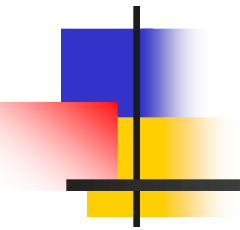
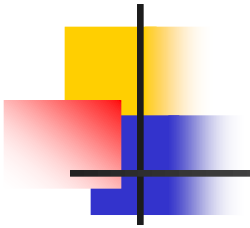


Chapter 2

ALGORITHM ANALYSIS

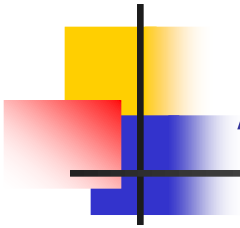


Objective

- To introduce:
  - Concept and basic terminologies
  - Algorithms analysis

CONTENT

- Algorithms
- Algorithms analysis
  - Algorithm Complexity
  - Big-O Notation



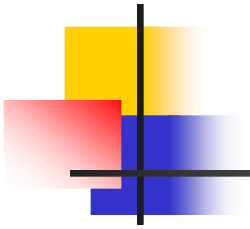
Algorithms

- **Algorithm:** a set of well-defined rules for a solution of a problem.
- An algorithm takes the input to a problem and transforms it into the output that solve the problems.
  - E.g. Problem : unsorted characters
  - Input : DCBA
  - Output : ABCD
  - Algorithm: [how to sort?]
- A problem can have many algorithms.

Algorithm

- Definition: steps to solve a problem.
  - Input → Process → Output
- Data structure + Algorithm → program
- Example 1:
  - To find a summation of N numbers in an array of A.

```
1 Set sum = 0
2 for j = 0 to N-1 do :
3 begin
        sum = sum + A[j]
4 end
```



Algorithm

- Example 2:
  - To find a summation of N numbers in an array of A.

```
1 sum = 0
2 for j = 0 to N-1
        sum = sum + A[j]
```

Algorithm

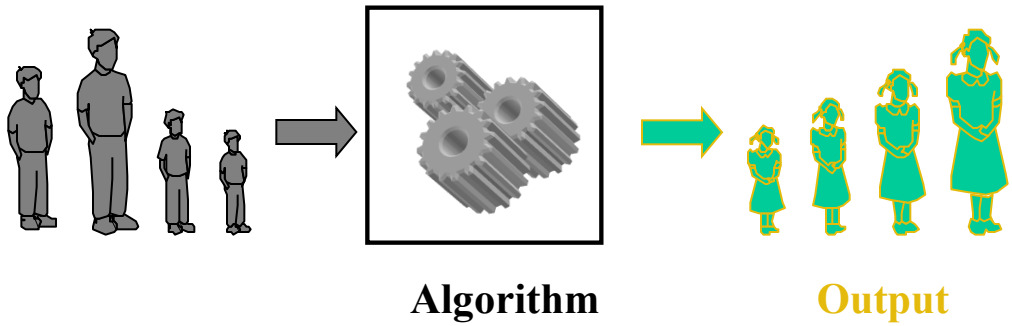
- Example 3:
  - Multiply (Matrix a, Matrix b)

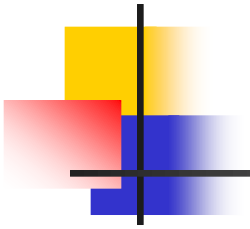
1. If column size of a equal to row size of b.
2. for i = 1 to row size of a
   - 2.1 for j = 1 to column size of b
     - $c_{ij} = 0$
   - 2.2 for k = 1 to column size of a
     - $c_{ij} = c_{ij} + a_{ik} * b_{kj}$

Algorithms properties

An algorithm possesses the following properties:

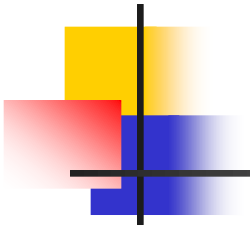
- It must be correct.
- It must be composed of a series of concrete steps.
- It must be composed of a finite number of steps.
- It must terminate (for all inputs).





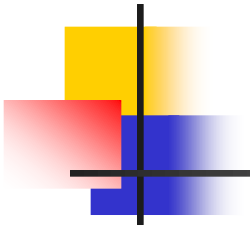
Algorithm Analysis

While analyzing a particular algorithm, we usually count the **number of operations performed by the algorithm.** We focus on the number of operations, not on the actual computer time to execute the algorithm. This is due to the fact that a particular algorithm can be implemented on a variety of computers and the speed of the computer can effect the execution time. However, the number of operations performed by the algorithm would be the same on each computer.



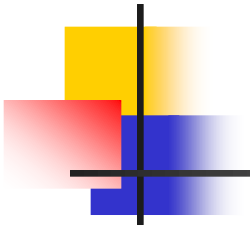
Algorithm Analysis

- To determine how long or how much spaces are required by that algorithm to solve the same problem.
- Other measurements:
  - Effectiveness
  - Correctness
  - Termination
- Effectiveness
  - Easy to understand
  - Easy to perform tracing
  - The steps of logical execution are well organized.



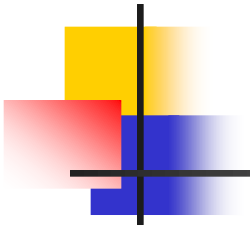
Algorithm Analysis

- Correctness
  - Output is as expected or required and correct.
- Termination
  - Step of executions contains ending.
  - Termination will happen as being planned and not because of problems like *looping*, out of memory *or infinite value*.



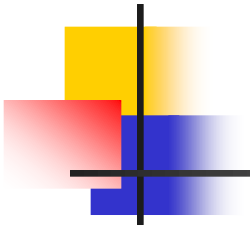
Algorithm Analysis

- Measurement of algorithm efficiency.
  - Running time
  - Memory usage
  - Correctness



Algorithm Complexity

- Algorithm *M* complexity is a function, *f*(*n*) where the running time and/or memory storage are required for input data of size *n*.
- In general, complexity refers to running time.
- If a program contains no loop, *f* depends on the number of statements. Else *f* depends on number of elements being process in the loop.



Algorithm Complexity

- Looping Functions can be categorized into 2 types:
  - Simple Loops
    * Linear Loops
    * Logarithmic Loops
  - Nested Loops
    * Linear Logarithmic
    * Dependent Quadratic
    * Quadratic

Algorithm Complexity

- Simple Loops
- 1. Linear Loops
  - Algo. 1a:

```
i = 1
loop (i <= 1000)
        application code
        i = i + 1
end loop
```

  - The number of iterations directly proportional to the loop factor (e.g. loop factor = **1000** times). The higher the factor, the higher the no. of loops.
  - The complexity of this loop proportional to no. of iterations. Determined by the formula: **f(n) = n**

Algorithm Complexity

- Simple Loops
- 1. Linear Loops
  - Algo. 1b:

```
i = 1
loop (i <= 1000)
        application code
        i = i + 2
end loop
```

  - Number of iterations half the loop factor (e.g. 1000/2 = **500** times).
  - Complexity is proportional to the half factor. **f(n) = n/2**
  - Both cases still consider Linear Loops - a straight line graphs.

Algorithm Complexity

- Linear Loops
- 2. Logarithmic Loops
  - Consider a loop which controlling variables - multiplied or divided:
  - Algo. 2a & 2b:

2a: Multiply Loops	2b: Divide Loops
1 i = 1	1 i = 1000
2 loop (i < 1000)	2 loop (i >=1)
1 application code	1 application code
2 i = i x 2	2 i = i / 2
3 end loop	3 end loop

Algorithm Complexity

Multiply		Divide	
Iteration	Value of i	Iteration	Value of i
1	1	1	1000
2	2	2	500
3	4	3	250
4	8	4	125
5	16	5	62
6	32	6	31
7	64	7	15
8	128	8	7
9	256	9	3
10	512	10	1
(exit)	1024	(exit)	0

Algorithm Complexity

- No of iterations is 10 in both cases.

  Reasons:

  → each iteration value of i double for multiply loops.

  → iteration is cut half for the divide loop.

- The above loop continues while the below condition is true:

  Multiply $2^{Iterations} < 1000$

  Divide $1000/2^{Iterations} \geq 1$

- Therefore the iterations in loops that multiply or divide are determined by the formula: **$f(n) = [\log_2 n]$**

Algorithm Complexity

- Nested Loops
  - Loops that contain loops :

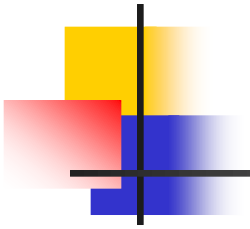
    Iterations= Outer loop iterations x Inner loop iterations

- 3. Linear Logarithmic
  - Algo. 3:

```
i = 1
        loop (i <= 10)
                j = 1
                loop (j <= 10)
                        application code      } inner loop
                        j = j * 2             }
                end loop
                i = i + 1
        end loop
```

outer loop

* Inner loop → Logarithmic loops ($f(n) = \log_2 n$)
* Outer loop → Linear loops ($f(n) = n$)



Algorithm Complexity

- f(n) = Outer Loop x Inner Loop = 10 * [$\log_2 10$]
- Generalized the formula as: **f(n) = [n $\log_2$ n]**

Algorithm Complexity

- 4. Dependent Quadratic
  - Algo. 4:

```
i = 1
loop (i <= 1000)
        j=1;
        loop (j <= i)
                application code      } inner loop
                j = j + 1            }
        end loop
        i = i + 1
end loop
```

outer loop

* Outer loop → Linear loops (f(n) = n)

Algorithm Complexity

- Inner loop depends on the outer loop, it is executed only once the first iteration, twice the second iteration…
- No. of iterations in body of inner loops, if n = 10,

  1 + 2 + 3 + … + 9 + 10 = 55

  average 5.5 = 55/10 times @ $= \left\lceil \frac{n+1}{2} \right\rceil$
- f(n) = Outer Loop x Inner Loop = n * $\left\lceil \frac{n+1}{2} \right\rceil$
- The formula for dependent quadratic, **f(n) = n** $\left\lceil \frac{n+1}{2} \right\rceil$

Algorithm Complexity

- 5. Quadratic loop
  - Algo. 5:

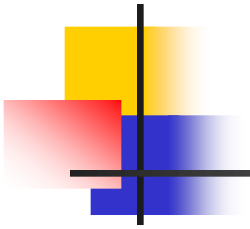
```
i = 1
        loop (i <= 10)                          ┐
                j = 1                           │
                loop (j <= 10)          ┐       │
                        application code│ inner │ outer
                        j = j + 1       │ loop  │  loop
                end loop                ┘       │
                i = i + 1                       │
        end loop                                ┘
```

* Inner loop → Linear Loops (f(n) = n)
* Outer loop → Linear Loops (f(n) = n)

f(n) = Outer Loop x Inner Loop = n * n

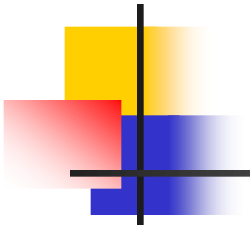
The generalized formula, **$f(n) = n^2$**

Algorithm Complexity:
Criteria of Measurement



- You could estimate the max time an algorithm could take – worst case time.
- If we can tolerate with worst-case time, the algorithm is acceptable.
- We could estimate the minimum or best-case time. If best-case if still too slow, we need another algorithm.
- More useful measure is average-case time but usually it is harder to find than the best and worst case, therefore we will find the worst-case time.

Algorithm Complexity:
Criteria of Measurement



- f(n) can be identified as:

  1. worse case: max value of $f(n)$ for any input

  2. average case: expected value of $f(n)$

  3. the best case: min value of $f(n)$

Algorithm Complexity:
Criteria of Measurement

- Example 1: Linear Searching

Worse case

Item is last element in an array or none.

$f(n) = n$

Average case

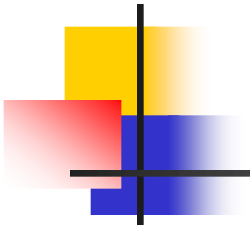
No item or in anywhere in an array location.

The number of comparison is any item in index 1, 2, 3,...,n.

Best case

Item is in the first position

$f(n) = 1$



Scenario

- After an algorithm is designed it should be analyzed.
- Usually, there are various ways to design a particular algorithm.
- Certain algorithm take very little computer time to execute, while others take a considerable amount of time.
- Consider the following problem. The holiday season is approaching and the gift shop is expecting sales to be double or even triple the regular amount. The shop has hired extra delivery persons to deliver packages on time. The company calculates the shortest distance from the shop to a particular destination and hands the route to the driver. Suppose that 50 packages are to be delivered to 50 different houses. The company, while creating the route, finds that 50 houses are one mile apart and are in the same area. The first house is also one mile from the shop (Figure 1).

Scenario

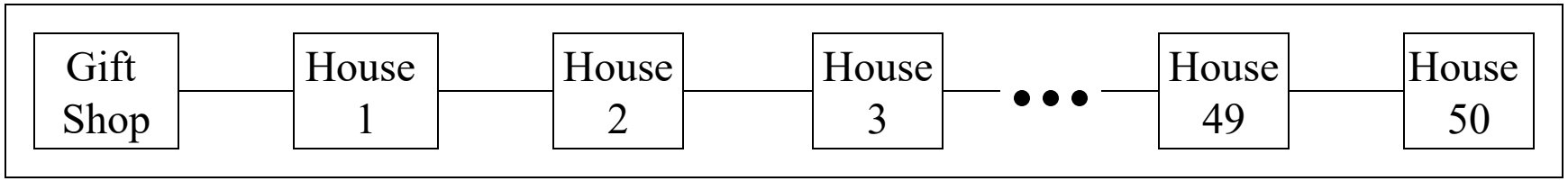


Figure 1: Gift Shop and the 50 houses

To simplify this figure, we use Figure 2:

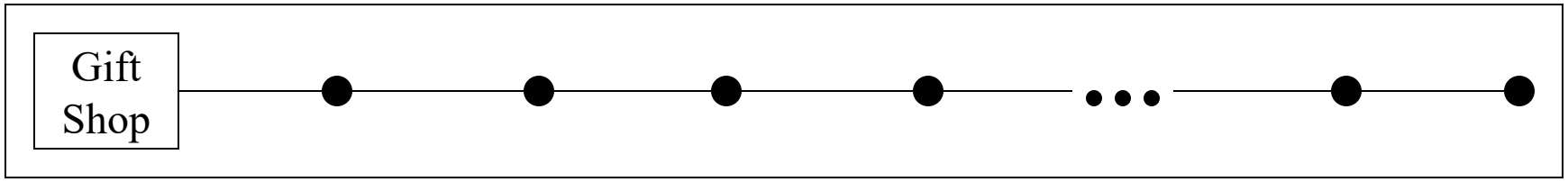


Figure 2: Gift shop and each dot representing a house

Each dot represents a house and the distance between houses, as shown in Figure 2, is 1 mile.

Scenario

To deliver 50 packages to their destinations, one of the drivers picks up all 50 packages, drives one mile to the first house, and delivers the first package. Then, he drives another mile and delivers the second package, drives another mile and delivers the third package, and so on. Figure 3 illustrates this delivery scheme.

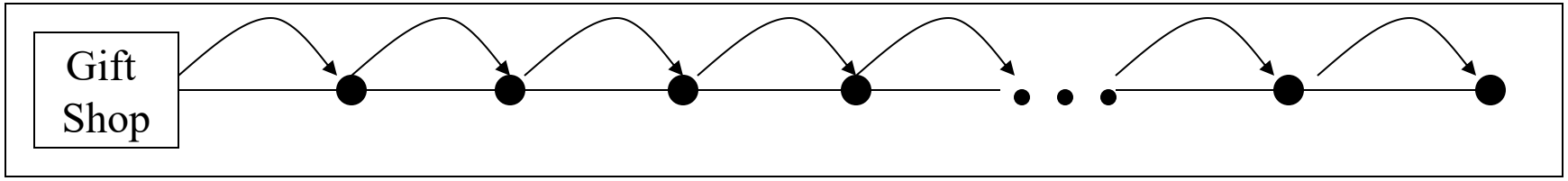


Figure 3: Package delivery scheme

Scenario

It follows that using this scheme, the distance the driver drives to deliver the packages is:

$1 + 1 + 1 + \ldots + 1 = 50$ miles

Therefore, the total distance travelled by the driver to deliver the packages and return to the shop is:

$50 + 50 = 100$ miles

Another driver has a similar route to deliver another set of 50 packages. The driver looks at the route and delivers the packages as follows: The driver picks up the first package, drives one mile to the first house, delivers the package, and then comes back to the shop. Next, the driver picks up the second package, drives 2 miles, delivers the second package, and then returns to the shop. The driver then picks up the third package, drives 3 miles, delivers the package, and comes back to the shop. Figure 4 illustrates this delivery scheme.

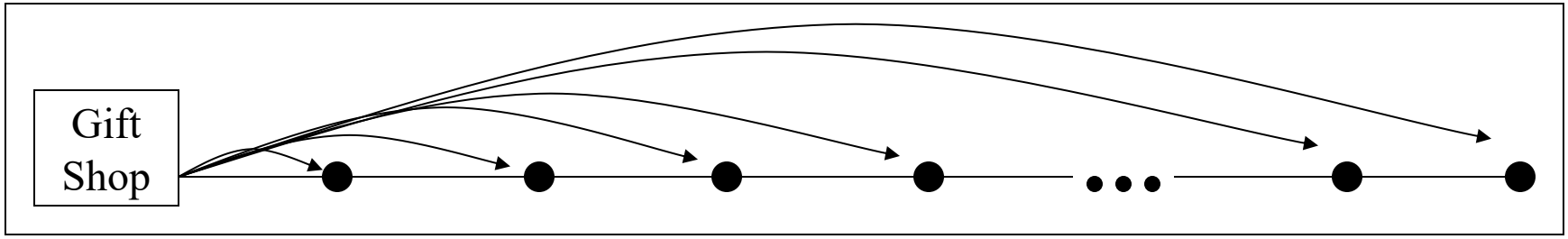


Figure 4: Another package delivery scheme

Scenario

The driver delivers only one package at a time. After delivering a package, the driver comes back to the shop to pick up and deliver the next package. Using this scheme, the total distance travelled by this driver to deliver the packages and return to the store is:

$$2 * (1 + 2 + 3 + \ldots + 50) = 2550 \text{ miles}$$

Now suppose that are n packages to be delivered to n houses, and each house is one mile apart from each other as shown in Figure 4. If the package are delivered using the first scheme, the following equation gives the total distance travelled:

$$\underbrace{1 + 1 + 1 + \ldots + 1}_{n \text{ times}} + n = 2n \qquad (1)$$

If the packages are delivered using the second method, the distance travelled is:

$$2 * (1 + 2 + 3 + \ldots + n) = 2 * (n(n + 1) / 2) = n^2 + n \qquad (2)$$

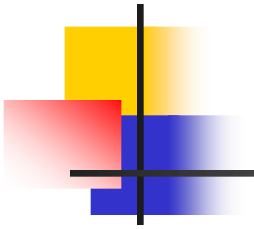
Scenario

In Equation (1), we say that the distance travelled is a function of n. Now consider Equation (2). In this equation, for large values of n, we find that the term consisting of n^2 becomes the dominant term and the term containing n is negligible. In this case, the distance travelled is a function of n^2. Table 1 evaluates Equations (1) and (2) for certain values of n. This table also shows the values of n, 2n, n^2, and $n^2 + n$.

Table 1: Values of n, 2n, n^2, and $n^2 + n$

n	2n	n^2	$n^2 + n$
1	2	1	2
10	20	100	110
100	200	10000	10100
1000	2000	1000000	1001000
10000	20000	100000000	100010000

Big-O Notation



- Order of magnitude of the result based-on run-time efficiency
- Expressed as O(n) @ O(f(n)) → big-O

  n – represents data, instructions, etc.
- Sometimes refer as <u>complexity degree of measurement</u>:

  * run-time ↓, complexity ↓

  * comparison of fast and slow algorithms in solving the same problem.
- Time taken= Algorithm Execution = Running Time

Big-O Notation

Table 2: Measures of Efficiency

Nested	Efficiency/ Complexity Degree	Algorithm Name	Algorithm Type
No Loop	O(k)	Constant	
Simple Loop	O(n)	Linear	Linear Search
	$O(\log_2 n)$	Logarithmic	Binary Search
Nested Loop	$O(n^2)$	Quadratic	Bubble Sort, Selection Sort, Insertion Sort
	$O(n \log_2 n)$	Linear Logarithmic	Merge Sort, Quick Sort, Heap Sort

Big-O Notation

Table 3: Intuitive interpretations of growth-rate function

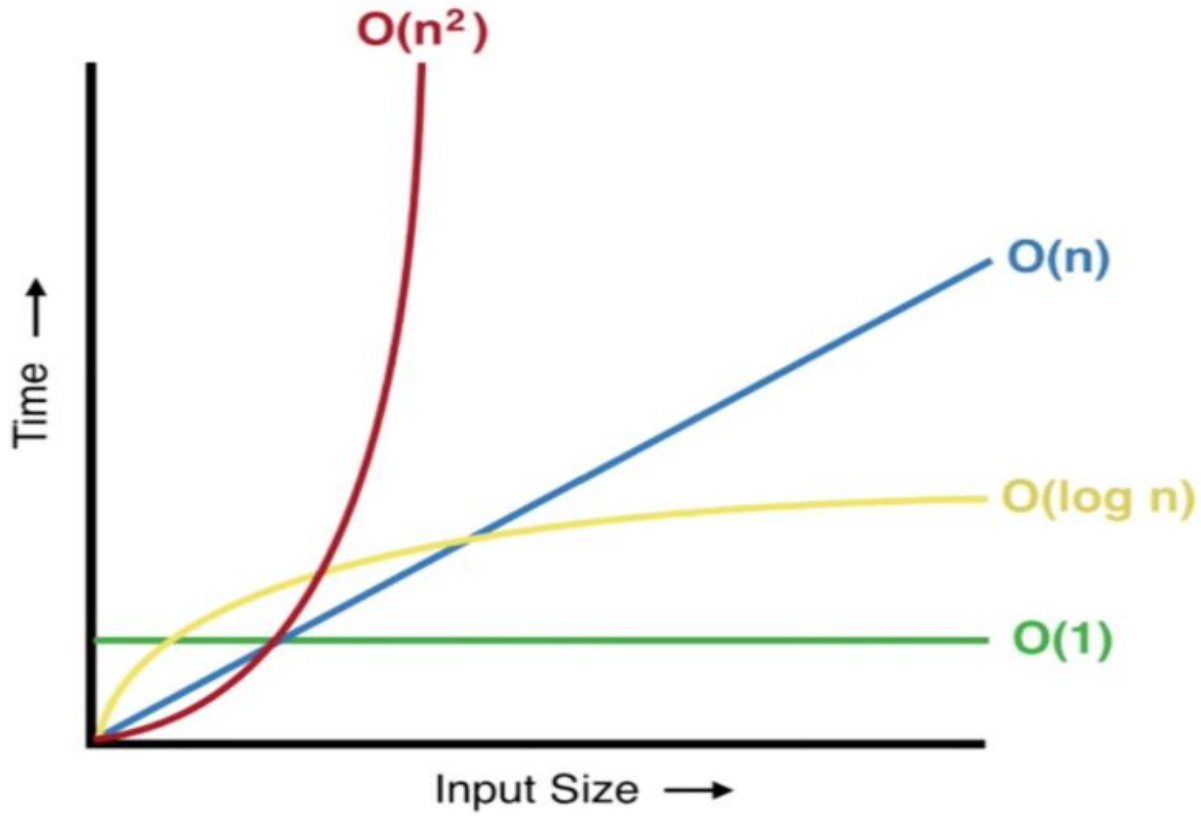
1	▪ A problem whose time requirement is constant and therefore, independent of the problem's size n.
$\log_2 n$	▪ The time for the logarithmic algorithm increases slowly as the problem size increases. ▪ If you square the problem, you only double its time requirement. ▪ The base of the log does not affect a logarithmic growth rate, you can omit it in a growth-rate function. ▪ Ex: recursive
n	▪ The time requirement for a linear algorithm increases directly with the size of the problem. ▪ If you square the problem, you also square its time requirement.

Big-O Notation

Table 3: Intuitive interpretations of growth-rate function

$n \log_2 n$	▪ The time requirement increases more rapidly than a linear algorithm. ▪ Such algorithms usually divide a problem into smaller problems that are each solved separately, ex : mergesort
n^2	▪ The time requirement increases rapidly with the size of the problem. ▪ Algorithm with 2 nested loop. ▪ Such algorithms are practical only for small problems.
n^3	▪ The time requirement increases rapidly with the size of the problem than the time requirement for a quadratic algorithm. ▪ 3 nested loop. ▪ Practical only for small problem.

Growth rate functions



Big-O Notation

- The big-O notation derived from f(n) using the following steps:
  1. Set the coefficient of the term to 1.
  2. Keep the largest term in the function and discard the others.
  (Ignore low-order terms and multiplication constant of higher order term.)
- Ex. 1:

$$f(n) = n * \left\lfloor \frac{n+1}{2} \right\rfloor$$

$$= \frac{1}{2}n^2 + \frac{1}{2}n$$

$$= n^{2+}n$$

$$= n^2$$

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

- Ex. 2:

$$f(n) = 8n^3 - 57n^2 + 832n - 248$$

$$= n^3 - n^2 + n - 1$$

$$= n^3$$

$$O(f(n)) = O(n^3)$$

Big-O Notation

- Ex. 3:

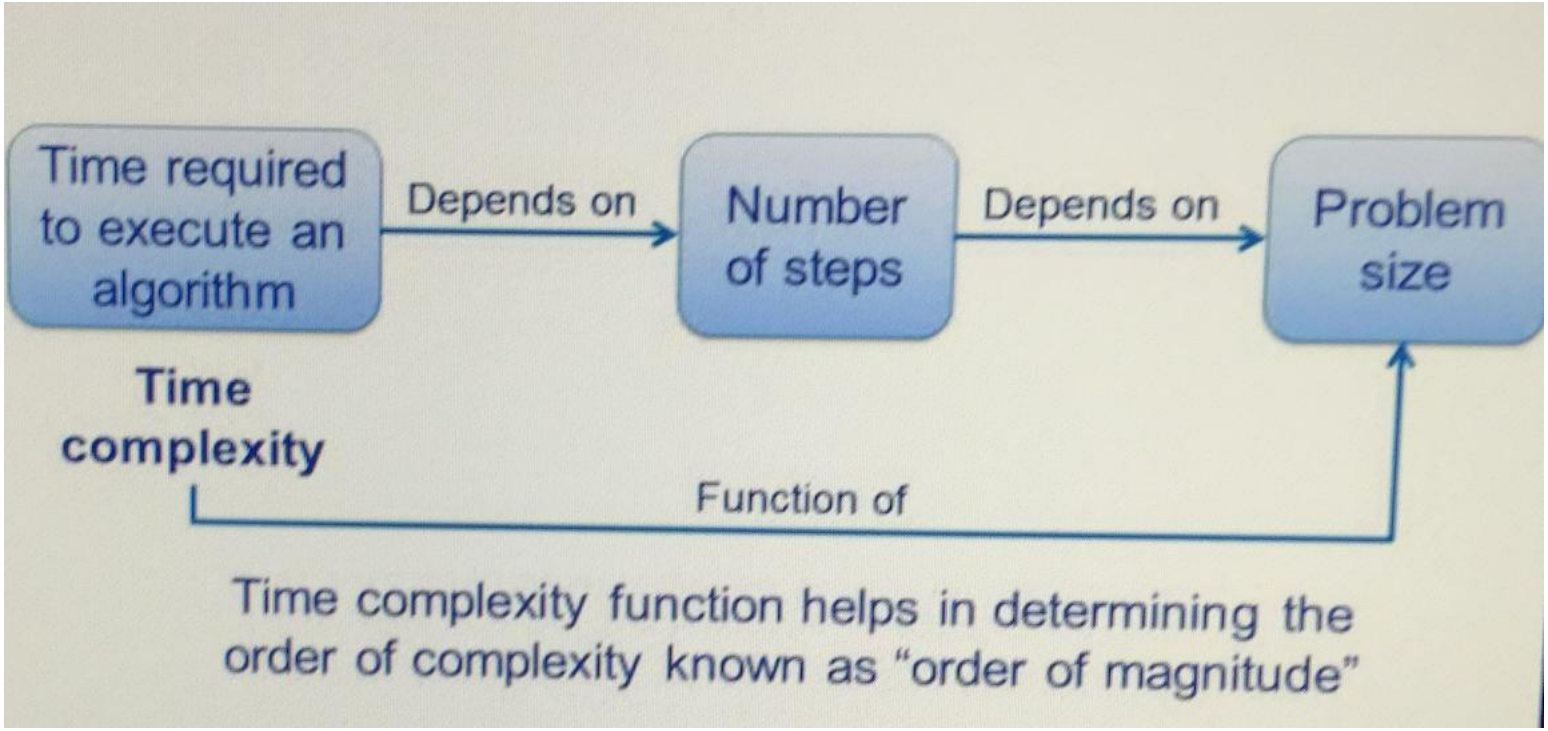
  Algorithm to calculate average
  1. initialize sum = 0
  2. initialize i = 0
  3. while i < n do the following :
     a. add x[i] to sum
     b. increment i by 1
  6. calculate and return mean

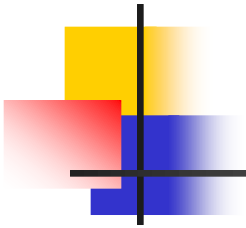
  1st Method

  f(n) = Linear loops
  = n

  Big-O → O(n)

2nd method:
Step Count Method





Big-O Notation

- Step Count Method – steps in an algorithm
- Each operation in an algorithm is a step
  - Mathematical operation
  - Logical operation
  - Conditional operation
  - Function calls
  - Assignment operation
  - Array reference

Big-O Notation

- 2nd Method

Statement	of time executed
1	1
2	1
3	n + 1
4	n
5	n
6	1
Total	3n + 4

$f(n) = 3n + 4$
$= n + 1$
$= n$
big-O → O(n)

Big-O Notation

- The comparison on run-time efficiency of different f(n) where n = 256 and 1 instruction → 1 microsec. (10^{-6} sec.)

f(n)	Estimation Time
n	0.25 milisec.
n^2	65 milisec.
n^3	17 sec.
$\log_2 n$	8 microsec.
$n \log_2 n$	2 milisec.

Note:

1 sec. → 1000 miliseconds
1 milisec. → 1000 microsec.

Big-O Notation

Example of calculation:

$f(n) = n$

$= 256 \times 10^{-6}$ sec.

$= (256 \times 10^{-6}) \times 10^{3}$ milisec.

$= 256 \times 10^{-3}$

$=$ **0.256 milisec.**

$f(n) = n^2$

$= 256^2$

$= 65536 \times 10^{-6}$ sec.

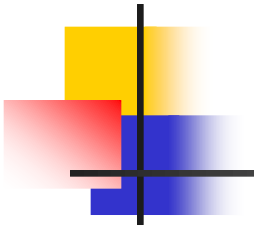
$= (65536 \times 10^{-6}) \times 10^{3}$ milisec.

$= 65536 \times 10^{-3}$

$=$ **65 milisec**.

- The run-time efficiency order of magnitude:

$(1) < O(\log_2 n) < O(n) < O(n\log_2 n) < O(n^2) < O(n^3)$



Exercise 1

1. Reorder the following efficiency from smallest to largest:
   a) 2^n
   b) n!
   c) n^5
   d) 10,000
   e) $n\log_2(n)$

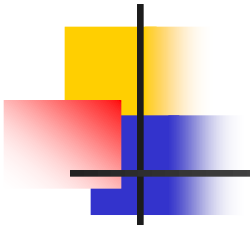
2. Reorder the following efficiency from smallest to largest:
   a) $n\log_2(n)$
   b) $n + n^2 + n^3$
   c) $n^{0.5}$

Exercise 1

3. Calculate the run-time efficiency for the following program segment: (doIT has an efficiency factor 5n).

```
1       i = 1
2       loop i < = n
            1       doIT(…)
            2       i = i + 1
3       end loop
```

4. Efficiency of an algo. is n^3, if a step in this algo. takes 1 nanosec. (10^{-9} sec.). How long does it take the algo. to process an input of size 1000?

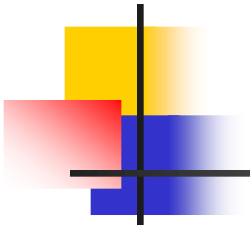


Exercise 1

5. Find the run-time efficiency of the following program segments

i)

```
// Calculate mean
        int sum = 0;
        double mean=0;
        for (int i=0; i  < n, i++) {
                sum += x[i];
        }
                mean = sum / n;
```



Exercise 1

ii)

```
// Matrix addition
for (int i = 0; i < n; i++)
for (int j = 0; j < n ; j++)
        c[i][j] = a[i][j] + b[i][j];
```