Data Structure
Chapter 1 & 3

Introduction and Algorithm Analysis

Dr. Patrick Chan
School of Computer Science and Engineering
South China University of Technology

Outline

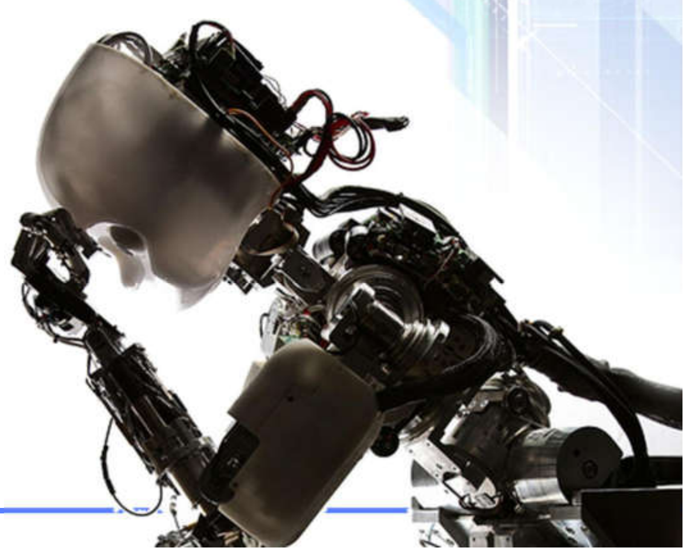
- Introduction (Ch1)
  - Philosophy of Data Structure (1.1)
  - Abstract Data Types and Data Structures (1.2)
  - Problems, Algorithms and Programs (1.3)

Outline

- Algorithm Analysis (Ch3)
  - Best, Worst and Average Cases (3.2)
  - Fast Computer or Fast Algorithm? (3.3)
  - Asymptotic Analysis (3.4)
  - Multiple Parameters (3.8)
  - Space Bounds (3.9)

Introduction: Efficient Programs?

- Program is the soul of a machine
- Objective of learning "**data structure**" is to improve the efficiency of a program
  - Shorter running time
  - Less memory



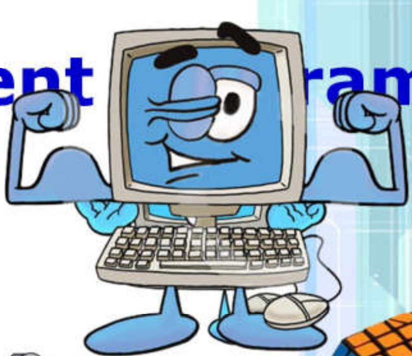
Introduction: Efficient Programs?

- 1982: Intel® 80286
  - 16 MHz (16,000,000 Hz)

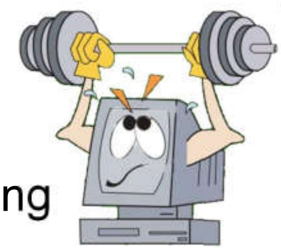
- 2023: Intel® Core™ i9-14900K
  - 6.00 GHz (6,000,000,000Hz)

- **Naïve Comparison:
  375 times difference!!**
  - Core i9 calculates **1 sec**
  - 80286 calculates **6.25 mins**
- Nowadays, the hardware is very powerful.
  Why do we still need to write an efficient program?

Introduction: Efficient Programs?



- **More powerful computers** encourage **more complex applications**
  - More calculations
  - Resources demanding



**With great power
comes great responsibility!**

- **A more efficient program is always desirable**

Data Structure

- **Data Structure** usually refers a complex data representation and its associated operations
  - e.g. Array of Integer | Insertion, Deletion
    Student Record | Update ID

Data Structure

- Proper data structure can make significant difference in program quality
- How to store an age?
  - Integer VS String
- How to find a minimum value from n integers?
  - Array VS Tree

Data Structure

- **Real Number** is better than **Integer**? **No**
- Every data structure has costs and benefits
  - No data structure is better than another in all situations
- A data structure requires:
  - **Space** for each data item it stores
  - **Time** to perform each basic operation
  - **Programming effort**

Data Structure

- Each problem has constraints, e.g. **time** and **space**
- **Data Structure Selection**:
  - **Analyze** the problem and determine the resource constraints
  - **Determine** the basic operations and quantify the resource constraints for each operation
  - **Select** the data structure that best meets these requirement

Detailed Definitions

- **Data:** a piece of information
  - e.g. 1
- **Type:** a collection of values
  - e.g. *Integer type*: collection of 1,2,3… value
- **Data Type:** a type and its related operations
  - e.g. *Integer data type*: Integer type and +-x÷ operations
- **Data Structure:** a complex type and its operations
- **Data Item:** a piece of information of a data type
  - e.g., 1 : a piece of information from a Integer type
  - A member of a data type

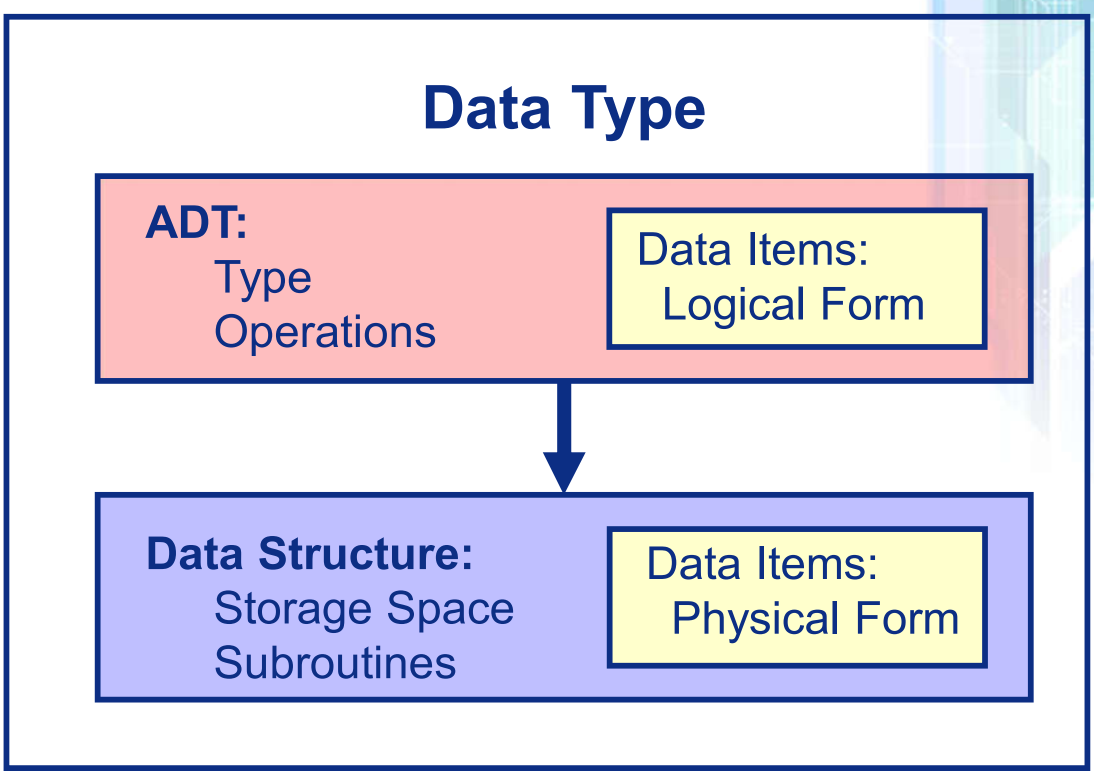
Detailed Definitions

- **Abstract Data Type:** a definition for a data type solely in terms of a set of values and a set of operations on that data type
  - ADT operation is defined by its inputs and outputs
  - Hide implementation details (Encapsulation)
- **Data Structure:** the physical implementation of an ADT
  - Operations associated with the ADT are implemented by subroutines (functions)
  - Usually refers to an organization for data in main memory

Logical vs Physical Form

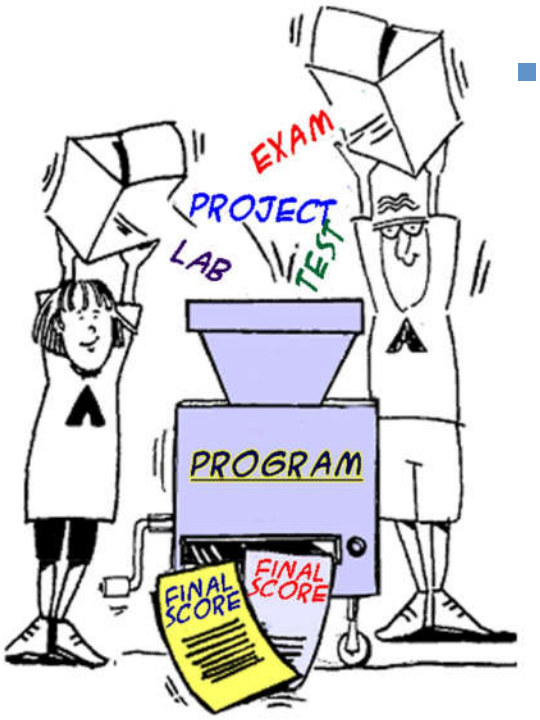
- Data items have both a logical and a physical form
  - **Logical Form**
    - Definition of the data item within an ADT
    - e.g. Integers in mathematical sense: +, -
  - **Physical Form**
    - Implementation of the data item within a data structure
    - e.g. 16/32 bit integers: overflow

Logical vs Physical Form



Detailed Definitions

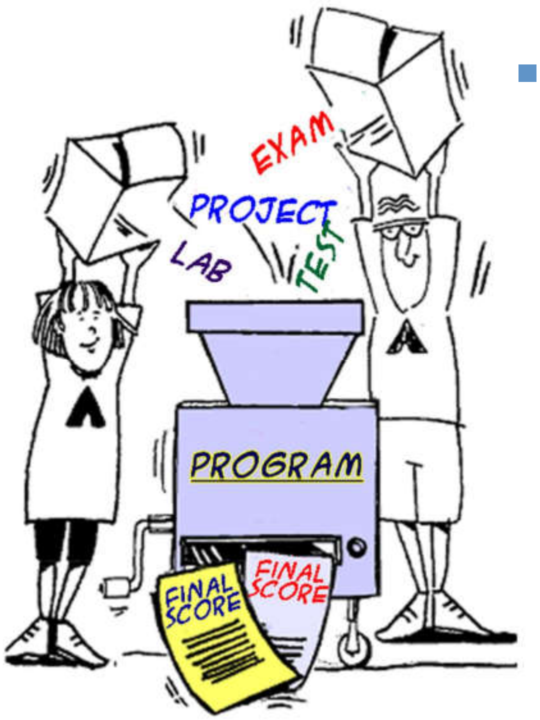
Problem, Algorithm & Program



- **Problem:** A task to be performed
  - Best thought of as inputs and matching outputs
  - Problem definition should include constraints on the resources that may be consumed by any acceptable solution

Detailed Definitions

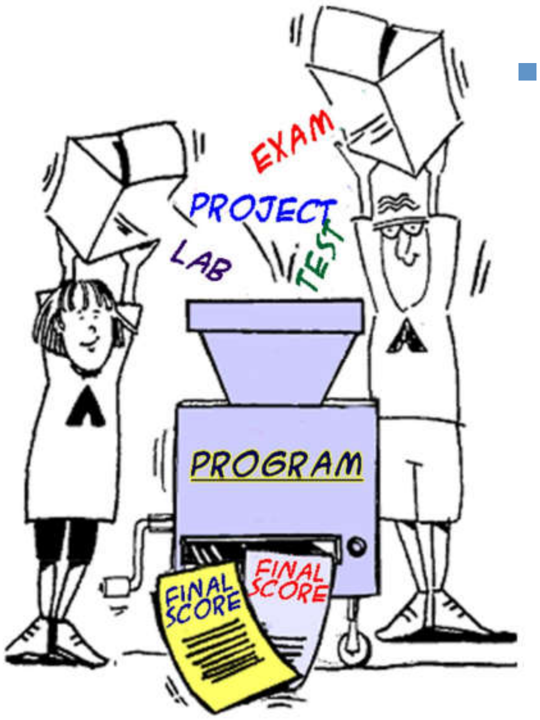
Problem, Algorithm & Program



- **Algorithm:** a method to solve a problem
  - Correct
  - No ambiguity
  - A series of concrete steps
  - A finite number of steps
  - Terminate

Detailed Definitions

Problem, Algorithm & Program



- **Program:** an instance for an algorithm in some programming languages

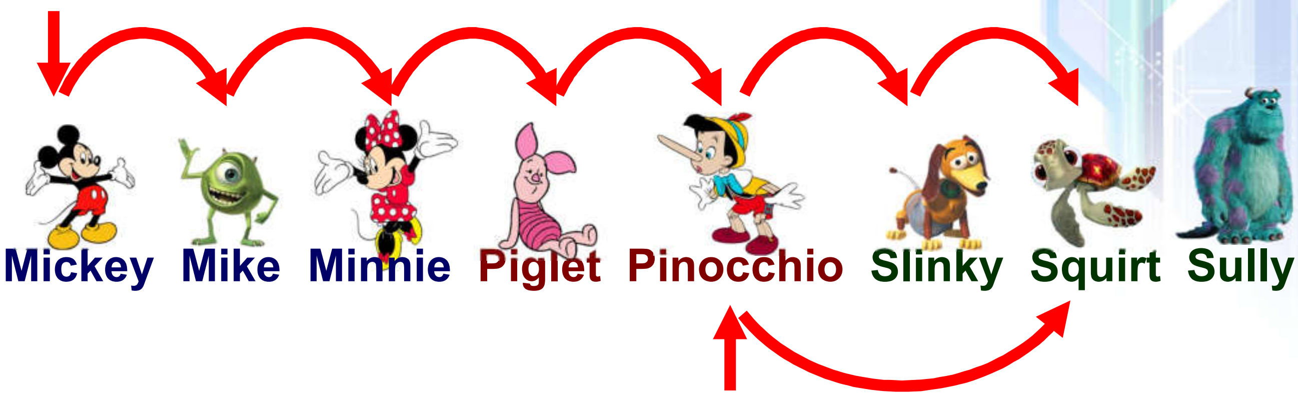
Algorithm Evaluation

- Many approaches (algorithms) to solve a problem. Which one is the best?
- Two criteria:
  - **Efficiency**
    - Concern of Data Structures and Algorithm Analysis
  - **Easy to understand**
    - Concern of Software Engineering
  - They are **conflicting**

Algorithm Evaluation

- Example: How can we find Squirt?

Sequential Search **Low Efficiency** **Easy**



Binary Search **High Efficiency** **Difficult**

How to Compare Efficiency?

- Two Methods:
  - **Empirical Comparison**
    - Run a program and see how much of the resources is used
    - Machine dependent
    - Difficult to setup and may be biased
  - **Algorithm Analysis**
    - Estimate the time and space needed for a program
    - Machine independent

Running Time T(n)

- Critical resource of a program is most often its **Running Time**
- Which one has the longest running time?
  - Assume n is input by users

```
a = n;
b = n*n;
c = n*n*n;
d = n*n*n*n;
ans = a+b+c+d;
```

```
ans = 0;
for i = 1 to 3
    ans = ans+n;
end
```

```
ans = 0;
for i = 1 to n
    ans = ans+i;
end
```

Running time does not rely on user's input

Running time relies on user's input

Running Time T(n)

- Focus on the program which depends on "size" of inputs
- $\mathrm{T}(n)$ for some **function** T on **input size** n

```
sum = 0;
```

$$\mathrm{T}(n) = c$$

c : the running time to assign a value to a variable

```
sum = 0;
for i = 1 to n
  for j = 1 to n
    sum = sum + 1;
  end
end
```

Loop n times

Loop n times

This c is not really important (not relate to n)

$$\mathrm{T}(n) = c + cn^2$$

c : the running time to assign a value to a variable

Algorithm Analysis
Running Time T(n): Exercise

```
s1 = 1;
s2 = 1;
s3 = 1;
```
c

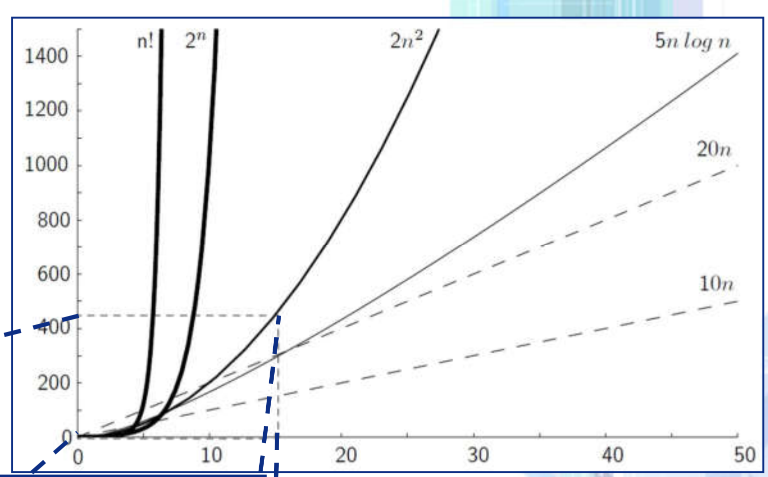
```
for a = 1 to n/2
  s1 = s1 + s1;
end
```
$cn/2$

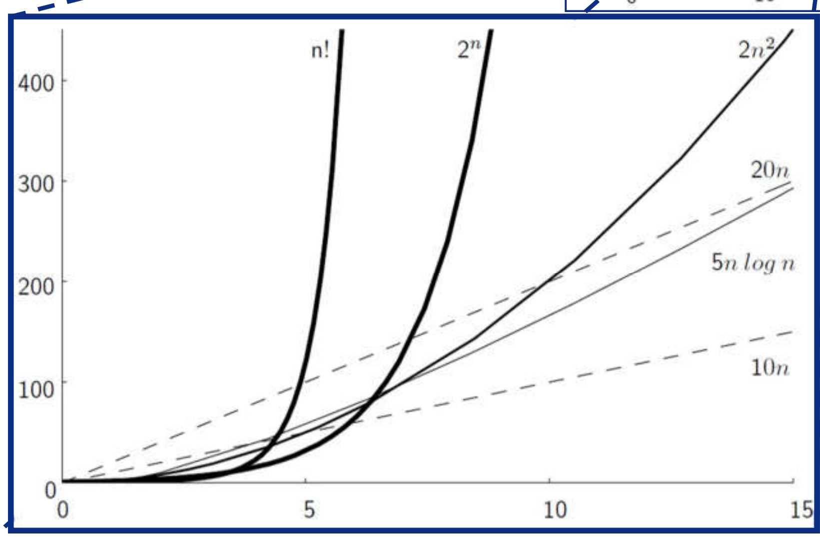
```
for b = 1 to n*n
  s2 = s2 * s2;
end
```
cn^2

```
for i = 1 to n
  for j = 1 to log(n)
    s3 = s1+s2+s3;
  end
end
```
$cn \log n$

$$T(n) = c + cn/2 + cn^2 + c\, n \log n$$

Algorithm Analysis
Growth Rate

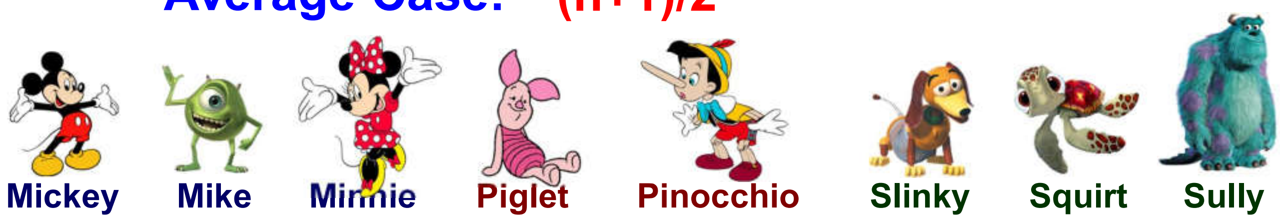




X-axis: input size
Y-axis: any measure of cost (e.g. time or space)

Algorithm Analysis
Best, Worst, Average Cases

- Same input size may require different amounts of running time
  - For example:
    Sequential search for K in an array of n integers
    - Begin at first element in array and look at each element in turn until K is found
    - **Best Case: 1**
    - **Worst Case: n**
    - **Average Case: (n+1)/2**



Algorithm Analysis
Best, Worst, Average Cases

- Which measure should be used?
  - **Best case**
    - May happen rarely
    - Too optimistic
  - **Worst case**
    - Upper bound
    - Important to real time algorithms
  - **Average case**
    - The fairest measure
    - Difficult to determine
      - Need to know the all possible inputs and their costs

Algorithm Analysis: Running Time T(n)

Fast Computer or Fast Algorithm?

- If we want to reduce the running time of a program, what should we do?
  - Buy **a faster computer**?
  - Write **a faster algorithm**?

Algorithm Analysis: Running Time T(n)

Fast Computer or Fast Algorithm?

- **Old Computer** (10,000 code/hour)
- **New Computer** (100,000 code/hour)

Size of input that can be processed using **OLD Computer** in one hour

Size of input that can be processed using **NEW Computer** in one hour

$T(n)$	n_{old}	n_{new}	Change	n_{new} / n_{old}
$10n$	1,000	10,000	$n_{new} = 10n_{old}$	10
$20n$	500	5,000	$n_{new} = 10n_{old}$	10
$2n^2$	70	223	$n_{new} = \sqrt{10}n_{old}$	3.16
2^n	13	16	$n_{new} = n_{old} + 3$	~1
$5n \log n$	250	1,842	$\sqrt{10}\, n_{old} < n_{new} < 10\, n_{old}$	7.37

Algorithm Analysis

- Which program has a lower time complexity?
  - **Program A:** $T(n) = cn^4$
  - **Program B:** $T(n) = cn + cn^2 + c \log n + cn^3$
- It is difficult to compare as there are many terms

Algorithm Analysis

- We would like to know what **the change of the complexity** is when n grows to ∞
- Three different measures:
  - **Big-Oh (O)**
  - **Big-Omega (Ω)**
  - **Big-Theta (Θ)**

Algorithm Analysis: Big-Oh

- Indicates **the upper bound** of a growth rate
- **Definition**
  For $\mathrm{T}(n)$ a non-negatively valued function,
  $\mathrm{T}(n)$ is in the set $\mathrm{O}(f(n))$
  if there exist two positive constants c and n_0
  such that $\mathrm{T}(n) \le c\,f(n)$ for all $n > n_0$
  - n_0 is the smallest value of n for which the claim of an upper bound holds true
  - Actually value of c is irrelevant

Algorithm Analy

if $\mathrm{T}(n) \le c\,f(n)$ for all $n > n_0$,
$\mathrm{T}(n)$ is in the set $\mathrm{O}(f(n))$

$\mathrm{T}(n) = 3n^2$

$$\mathrm{T}(n) \le c\,f(n)$$

$$3n^2 \le c\,f(n)$$

By substituting,

$c = 3 \quad f(n) = n^2 \quad n_0 = 1$

$$3n^2 = 3n^2$$

$\mathrm{T}(n)$ is in $\mathrm{O}(n^2)$

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

$$\mathrm{T}(n) \le c\,f(n)$$

$$3n^2 + n \le 3n^2 + n^2 = 4n^2$$

By substituting,

$c = 4 \quad f(n) = n^2 \quad n_0 = 1$

$$3n^2 + n \le 4n^2$$

$\mathrm{T}(n)$ is in $\mathrm{O}(n^2)$

Algorithm Analysis: Big-Oh
Exercise

if $T(n) \le c\, f(n)$ for all $n > n_0$, $T(n)$ is in the set $O(f(n))$

$$T(n) = c + cn/2 + cn^2 + cn \log n$$

$$T(n) \le s\, f(n)$$

n > log n, as n → ∞

$$c + cn/2 + cn^2 + cn \log n \le cn^2 + cn^2/2 + cn^2 + cn^2$$
$$= (c + c/2 + c + c)n^2$$
$$= (\ 7c/2\ )n^2$$

By substituting,

$s = 7c/2$ $\quad f(n) = n^2$ $\quad n_0 = 1$

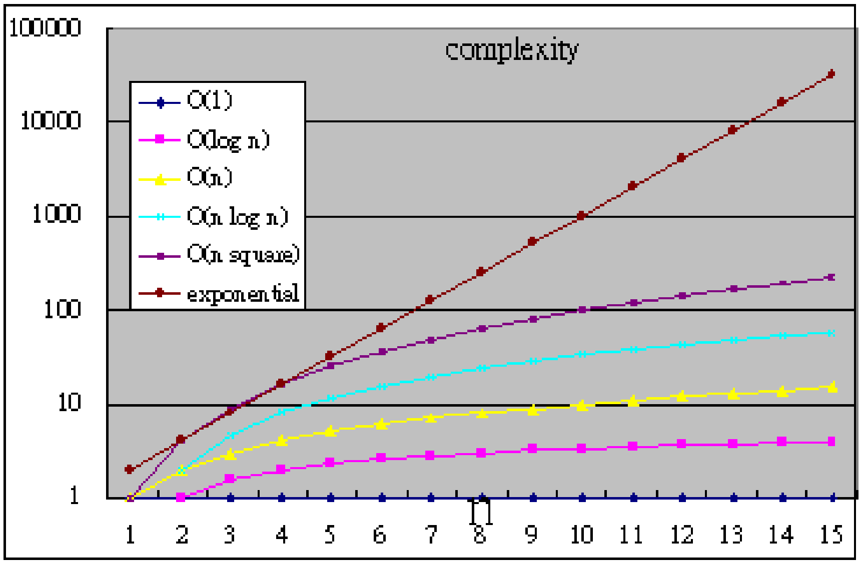
$$c + cn/2 + cn^2 + cn \log n \le (\ 7c/2\ )n^2$$

$T(n)$ is in $O(n^2)$

Algorithm Analysis: Big-Oh

- Given $T(n) = 3n$
- We know that "$T(n) = 3n$ is in $\mathbf{O(n)}$"
- Can we say "$T(n) = 3n$ is in $\mathbf{O(n^3)}$"?
- **Yes** but the **tightest** upper bound is preferred

Algorithm Analysis: Big-Oh



$O(1)$:	constant	$O(n\log_2 n)$:	n log n
$O(\log 2^n)$:	logarithmic	$O(n^2)$:	quadratic
$O(\log_2 2^n)$:	log squared	$O(n^3)$:	cubic
$O(n)$:	linear	$O(2^n)$:	exponential

Algorithm Analysis: Big-Oh

- **Big-Oh VS Worst Case**
  - Big-Oh refers to a growth rate
  - Worst case refers to the worst input from among the choices for possible inputs of a given size
  - e.g. Sequential Search
    - Big-oh: $T(n)$ is in $O(n)$
    - Worst Case: n

Algorithm Analysis: Big-Omega

- Indicates **the lower bound** of a growth rate
- **Definition**
  For $\mathrm{T}(n)$ a non-negatively valued function,
  $\mathrm{T}(n)$ is in the set $\Omega(g(n))$
  if there exist two positive constants c and n_0
  such that $\mathrm{T}(n) \geq c\, g(n)$ for all $n > n_0$
  - n_0 is the smallest value of n for which the claim of an upper bound holds true
  - The actually value of c is irrelevant

Algorithm Analy

if $\mathrm{T}(n) \geq c\, f(n)$ for all $n > n_0$,
$\mathrm{T}(n)$ is in the set $\Omega(f(n))$

$\mathbf{T}(n) = 3n^2$

$$\mathrm{T}(n) \geq c\, f(n)$$

$$3n^2 \geq c\, f(n)$$

By substituting,

$c = 3 \quad f(n) = n^2 \quad n_0 = 1$

$$3n^2 = 3n^2$$

T(*n*) is in Ω(*n*²)

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

$$\mathrm{T}(n) \geq c\, f(n)$$

$$3n^2 + n \geq 3n^2$$
$$= 3n^2$$

By substituting,

$c = 3 \quad f(n) = n^2 \quad n_0 = 1$

$$3n^2 + n \geq 3n^2$$

T(*n*) is in Ω(*n*²)

Algorithm Analysis: Big-Theta

- When O and Ω are the same, we indicate this situation by using Θ notation
- **Definition**
  An algorithm is said to be $\Theta(h(n))$
  if it is in $O(h(n))$ and $\Omega(h(n))$

Algorithm Analysis: Big-Theta

- **Example 1**

```
a = b;
```

  - $T(n) =$

- **Example 2:**

```
sum = 0;
for (i=1; i<=n; i++)
  sum += n;
```

  - $T(n) =$

☺ Small Exercise ☺

- What is the Big-O, Big-Omega and Big-Theta of the following program?

```
sum = 0;
for (i=1; i<=n; i++)    //first loop
  for (j=1; j<=i; j++) //double loop
    sum++;
for (k=0; k<n; k++)     //second loop
  A[k] = k;
```

- $\mathrm{T}(n) = c_1 + c_2 \sum_{i=1}^{n} i + c_3 n \quad = c_1 + c_2 \dfrac{n(n+1)}{2} + c_3 n$
- $\mathrm{O}(n^2)$, $\Omega(n^2)$, $\Theta(n^2)$

☺ Small Exercise ☺

- What is the Big-O, Big-Omega and Big-Theta of the following program?

```
sum1 = 0;
for(i=1;i<=n;i++)    //first double loop
  for(j=1;j<=n;j++) //do n times
    sum1++;
sum2 = 0;
for(i=1;i<=n;i++)    //second double loop
  for(j=1;j<=i;j++) //do i times
    sum2++;
```

- $\mathrm{T}(n) = c_1 + c_2 n^2 + c_1 + c_2\, n(n+1)/2$
- $\Omega(n^2)$, $\mathrm{O}(n^2)$, $\Theta(n^2)$

Algorithm Analysis: Case Study
Binary Search

- How many elements are examined in worst case?

```
// Return position of element in sorted
// array of size n with value K.
int binary(int array[], int n, int K) {
  int l = -1;
  int r = n;          // l, r are beyond array bounds
  while (l+1 != r) {     // Stop when l, r meet
    int i = (l+r)/2;     // Check middle
    if (K < array[i]) r = i;      // Left half
    if (K == array[i]) return i;  // Found it
    if (K > array[i]) l = i;      // Right half
  }
  return n; // Search value not in array
}
```

Find: 45

Position	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Value	11	13	21	26	29	36	40	41	45	51	54	56	65	72	77	83

Algorithm Analysis: Case Study
Binary Search

- How many elements are examined in worst case?

Position	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Value	11	13	21	26	29	36	40	41	45	51	54	56	65	72	77	83

$$T(n) = T(n/2) + 1$$

where $n>1$ and $T(1) = 1$

- Therefore, $T(n) = \log_2 n + 1$
- Cost is $\Theta(\log_2 n)$

Algorithm Analysis
Other Control Statements

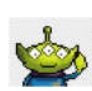
- **While loop**
  Analyze like a for loop
- **If statement**
  Take greater complexity of then/else clauses
- **Switch statement**
  Take complexity of most expensive case
- **Subroutine call**
  Complexity of the subroutine

Analyzing Problems
Multiple Parameters

- Compute the rank ordering for all **C** pixel values in a picture of **P** pixels.

```
for (i=0; i<C; i++)   // Initialize count
  count[i] = 0;
for (i=0; i<P; i++)   // Look at all pixels
  count[value(i)]++;  // Increment count
bubbleSort(count);    // Sort pixel counts
```

- $T(P, C) = C + P + C^2$
- $O(P + C^2)$
- $\Theta(P + C^2)$



C: 256 colors (8 bits)
P: 50 x 40 pixels

Space/Time Tradeoff Principle

- One can often reduce time if one is willing to sacrifice space, or vice versa, e.g.
  - Encoding or packing information
    - Boolean Flags
      - Boolean takes one bit, but a byte is the smallest storage, so pack 8 Booleans into 1 byte
  - Table lookup
    - Factorials
      - Compute once, store results, use many times